4. PRONG - Processing CSCI

4.1 CSCI Overview

The Processing CSCI is responsible for the initiation, managing, and monitoring of the execution of science software algorithms. These science software algorithms are identified to the Processing CSCI through a PGE. From the ECS Glossary, a PGE is defined as "a set of one or more compiled binary executables and/or command language scripts; it is the smallest unit that can be scheduled for the Product Generation System (PGS, now the Planning and Data Processing Subsystems) processing." A PGE is equivalent to one processing job which requires the use of the Data Processing Subsystem's hardware and software resources. Generally, a PGE will be used for the generation of ECS Data Products, but a PGE may be defined to perform other types of processing, such as pre-processing of input data or the quality assurance processing of generated Data Products. PGEs that generate data products and perform quality assurance are provided by the Instrument Teams and algorithm developers. The Processing CSCI is informed of the required execution of a PGE through a Data Processing Request message received from the Planning CSCI. The Processing CSCI will not initiate the execution of a PGE until all necessary data required as an input to the PGE is available. Available in this context means that the data exists at a Data Server, not necessarily the on-site Data Server.

In support of the execution of the science software algorithms, the Processing CSCI has the following responsibilities:

a. Manages the science software algorithm execution process.
b. Manages Science Data Processing computer hardware resources efficiently.
c. Manages the flow of data required to execute a science software algorithm.
d. Manages the flow of data produced by the execution of a science software algorithm.
e. Provides an Operational interface to allow monitoring of processing status, and manual intervention, when necessary, into Science Data Processing operations environment, including processing queue control.
f. Provides an Operational interface to support the quality assurance of generated data products.

4.1.1 Processing CSCI Design Rationale

The Processing CSCI detailed design, as presented, is meant to provide a framework to build a robust science data processing system. This detailed design has taken the concepts expressed in the Data Processing Subsystem sections of the SDPS System Design Specification and the SDPS Preliminary Design Specification and expanded, improved, and modified these concepts to reflect the current state of the Processing CSCI design.

This detailed design represents a significant step in providing a basis to continue into the Implementation phase. By developing the use-case scenarios (see Section 4.5, Processing CSCI Dynamic Model), the design has established the boundaries of the Processing CSCI software, allowed the identification of problem areas, and the resolution of these problem areas. These scenarios will now be used as guidance for implementation.

The Processing CSCI design rationale has been influenced by a set of design drivers. These design drivers have greatly influenced the decisions which have been made as the design has matured since the System Design Specification and the Preliminary Design Specification for the ECS Project. The following design drivers have been identified:

a. Separation of Planning and Processing—The roles and responsibilities of the Planning CSCI and Processing CSCI have been separated to support future evolving aspects of these functions.

b. Data Driven System—The Processing CSCI does not activate a Data Processing Request which requires execution until data is available to support the execution of the PGE associated with Data Processing Request.

c. Priority Driven Data Processing—As a Data Processing Request is input into the Processing CSCI, this Data Processing Request has an assigned priority which is determined in the Planning CSCI. This priority is based on the production rules used to generate the production plan which is currently active. This priority is reflected by the production policies at a DAAC.

d. Development of user interfaces—The amount of support to provide the different classes of users, i.e., Operations, Instrument Teams, and remote users, has influenced how information is to be provided to each type of user. The need to support a secure operations environment is quite evident.

e. Exception handling—The capabilities to support recovery from the faults associated with the failure of a resource or a PGE are provided to support efficiency in the science data processing environment.

f. Extent of automation vs. manual supported operations—To support an efficient Science Data Processing environment, a number of decisions have been made in the design to reflect the need to increase the amount of automated decision making required. This is driven by the need to support the generation of data products in an attended (by Operations staff) or unattended mode.

g. Dynamic aspects of Planning and Processing—Both Planning and Processing software must have the capability to react to real-time events, i.e., PGE failures, resource failures, etc., which have affected the active production plan and the queue of Data Processing Request Jobs awaiting execution in Processing.

Each of these design drivers has impacted the framework of the Processing CSCI design. To emphasize their impact in the different functional areas affecting the design, a description of the design in certain areas and the underlying factors affecting the design has been provided. These descriptions are in the following sections:

a. The division of the Planning and Processing CSCIs

b. Resource Management

c. Quality Assurance

d. Processing Error Architecture

e. Processing/Planning Interface

f. Processing/MSS Interface

g. Processing/Data Server Interface

The information provided in the following sections on the above listed topics is meant to summarize important aspects of the Processing CSCI detailed design. More information on these areas and the resulting impact to the detailed design is contained in the Section 4.5, Processing CSCI Dynamic Model, which contains a set of scenarios used to define the roles and responsibilities of the Processing CSCI.

4.1.1.1 The Division of the Planning and Processing CSCIs

The division of the Planning and Processing CSCIs was influenced by the following factors:

a. Follows Client/Server Architecture—Planning acts as the Client, and Processing acts as the Server. Planning is responsible for developing a Production plan and informing Processing which activities must be executed as listed in the Production plan.

b. Increases Fault Tolerance of the Planning and Data Processing Subsystems—By separating the Planning services from the Processing services, an increase in the fault tolerance of the Planning and Data Processing Subsystem software is achieved. The failure of the Planning Subsystem will not cause an immediate breakdown in production processing. Processing will be able to continue with production processing until the processing queues are depleted. Also, a failure of the Processing CSCI or Processing Hardware will not cause the Planning Subsystem to fail. Although the flow of information from Planning to Processing and Processing to Planning will be interrupted, this should not affect the overall production plan.

c. Evolvability of the roles and responsibilities of Planning as ECS becomes operational—The separation of Planning services from Processing services allows for different configurations of Planning services and Processing services. For example, there may be some special event which would result in the Planning services of a DAAC creating a Production plan for itself as well as other DAACs. By separating Planning from Processing, this becomes an easier problem to resolve.

Each of the above items has influenced the separation of Planning services from Processing services. This separation will support the general concept of the changing roles of Planning and Processing as ECS matures.

4.1.1.2 Resource Management

The Resource Management capabilities as presented in the Processing CSCI detailed design have been influenced by the following design drivers:

a. Efficient use of computer resources is required to support Production Processing—The Processing CSCI is responsible for the allocation of resources, i.e., disk space, memory, and CPU, to execute a PGE. To perform this role, Processing relies on MSS to provide up to date (pseudo-real-time) information on the health and overall availability of Data Processing Hardware resources. This information is provided by MSS SNMP monitoring agents which will be located on each science processing resource platform.

   The Processing CSCI allocates disk space, memory, and CPU to support the execution of a PGE. The PGE resource profile information for allocating disk space, memory and CPU are established during the AI&T time frame and will undergo updates as the science software matures in the science data processing environment

Before a PGE begins execution, all required resources must be available. This means that enough disk space and sufficient CPU must exist to support execution. These resources will be allocated to only support the execution of a given PGE. Also, as required through the Planning CSCI, all input data is available before initiating the PGE. This will alleviate any deadlock situation where a PGE is awaiting some input data file or awaiting the use of a given resource currently in use by another PGE. Also, during execution monitoring, Processing will monitor the use of the allocated disk space, memory, and CPU as a method of fault detection.

b. Automated recovery from resource faults—Processing plays a limited role in resource fault recovery. Processing is responsible for protecting the state data maintained in the Processing software and for insuring that the execution of a PGE can resume upon the recovery of a resource. When a resource has failed, the Processing CSCI will inform MSS of the failure and update the resource management information associated with the failed resource to indicate that the resource can not be allocated for processing. Also, MSS will inform Processing that a resource is unavailable and when the resource is available again for processing. This interface is intended to be two way.

c. Reporting of resource fault and other information—Processing supports fault reporting in two distinct ways:

   1. During PGE execution—While a PGE is executing, information is collected from the execution of the science software through the use of a Status Message File. Processing will inform MSS about resource-specific faults. To recover from a resource fault, the Processing CSCI may determine if the PGE can be executed on another resource which is capable of supporting the execution of the PGE. This will be done by allowing the Data Processing Request Jobs to re-queue into the queue supporting the other resource. For a science software fault, Processing will use return code information provided by the PGE to determine the necessary steps to take for recovery purposes.

   2. Processing SW specific—If an event for which Processing is responsible (i.e., data staging, data destaging, execution environment monitoring) indicates a possible resource-specific fault, Processing will inform MSS.

d. Logging of resource fault and other information—Resource fault and usage information will be logged for accounting, performance, and accountability purposes. The resource usage characteristics for a PGE will be updated and used for future Production planning purposes. Fault information will be used to determine if faults are re-occurring frequently. The logging of resource management data is performed by MSS. Processing will assist by providing system management information to MSS.

4.1.1.3 Quality Assurance

This section briefly describes quality assurance and the support provided by SDPS. Four types of quality assurance will be supported by SDPS.

a. In-Line QA. This service is provided through the production environment. This is the automated quality assurance processing that can be provided by a PGE executing a quality assurance algorithm which only performs automated quality assurance on a generated product. These quality assurance PGEs will be provided by Instrument Team algorithm developers.

b. DAAC QA. Two capabilities exist for performing QA at a DAAC: DAAC manual QA and DAAC Non-Science QA. Performing a cursory view of products after using the Data Server subscription service to request products to view is called DAAC manual QA. The automated process that evaluates the quality of the production process is called DAAC Non-Science QA.

   To visualize products requires submittal of a subscription request for products which is a manual process. Then when subscription notifications indicate that requested products are available they have to be retrieved from the Data Server. Thereafter, they can be viewed using a visualization COTS tool. There is also a COTS editor available for updating product metadata with a DAAC operator flag that indicates QA status of a product.

   DAAC Non-Science QA occurs during the post-processing stage of the production process. Before DPR data are destaged and resources deallocated DAAC Non-Science QA runs. It collects and reports information about PGE input and output granules and product metadata on a run-by-run basis.

c. SCF QA. This service is provided through use of the Data Server subscription service. A user who wishes to perform QA on a product can request a copy of the product whenever the product is generated by placing a subscription for the data product. Product QA is a process involving the use of COTS visualization and editting tools. These tools enable scientists to assist with the analysis of the quality of products and update product metadata with a SCF operator flag that indicates the QA status of a product.

By using the Data Server subscription service, the Processing CSCI lessens the dependency on man-in-the-loop activities affecting the generation of data products. Processing SW is being designed under the assumption that Data Processing must be able to generate products while in an attended or unattended mode. If the Q/A position is manned when the product is generated, the product can be reviewed while still on the Data Processing Subsystem storage device. Otherwise, the Q/A position may have to request the product from the Data Server.

The Processing CSCI will support an interface for visual display of a data product through the use of data visualization tools, such as EOSVIEW and IDL. Processing will also support an interface to allow for the update of the Q/A metadata which is associated with a data product.

The design approach was chosen for the following reasons:

a. Based on the Data Processing design approach of allowing for manned and unmanned modes of operations.

b. Follows Q/A approach for making products available for SCF review. No new paradigm to support DAAC manual Q/A has been introduced. This leads to maximum reuse of already developed software capabilities.

c. Allows man-in-the-loop Q/A activities at the DAAC without serious impact on the processing resources. This approach would not require manual intervention before processing of other data products continues. If Q/A position is unmanned, the data products requiring Q/A review are stored at the Data Server and await review.

4.1.1.4 Processing Error Architecture

This section provides a brief description of the Error Architecture approach adopted by the Processing CSCI and follows the common error handling approach adopted across all applications. Error Architecture refers to the mechanisms used for error detection, reporting and recovery that are incorporated into the design of the Processing CSCI. It provides details on how the Processing CSCI will react when an error or exception, i.e., hardware or software, occurs during steady-state processing of a Data Processing Request and the execution of a PGE.

This Processing CSCI Error Architecture approach was influenced by the following factors:

a. Detection of different types of errors, i.e., Science Software (PGE) execution errors, Processing Hardware errors, and Processing Software errors.

    1. The detection of science software errors which occur during the execution of a PGE are captured by the SDP Toolkit, and are propagated to the Processing CSCI through the use of a PGE return code status. The Processing CSCI will have knowledge of the error associated with a given return code, and as a result of this return code, may initiate a corrective measure, such as alerting the operations staff or restarting the PGE job with updated diagnostic flags to initiate the capture of detailed diagnostic information. The definition of these return codes and the resulting corrective measures are an ongoing activity. Dialogue with Release A instrument teams, i.e., CERES, have led to some initial definitions. These are currently being defined with more detail and will be supplied to all instrument teams for more feedback.

       As a result of an unsuccessful return code, the running of Data Processing Request jobs which were dependent on the PGE that failed would not be executed. This information would be made available to the Operations staff through the use of the Processing Operations Interface. Please see Section 4.1.3, COTS Selection, for more information on the Processing Operations Interface. The Processing Operations Interface is provided by the selected COTS product.

    2. Although MSS will also detect science processing hardware errors, the detection of Processing HW and Processing SW errors can occur in the Processing SW. Processing HW error detection will be supported by a group of MSS interfaces used to support fault tolerance and provide resource management information.

b. These errors must be reported to different user classes depending on the nature of the error, i.e., IT/Algorithm Developer, DAAC Operations personnel. The IT/Algorithm Developer users are interested in the detailed error information associated with the execution of a PGE for the purposes of debugging a problem. This information will be logged in Status Message Files which are created and maintained during PGE execution by the SDP Toolkit. While the DAAC Operations personnel is interested in whether a PGE was successfully executed, the detailed error information will not be provided for review, unless requested.

Rather, the DAAC Operations personnel will be alerted to the unsuccessful processing, and the alert information will be captured in the processing log for later use.

c. Recovery actions from errors will differ on the type of error. The recovery from an error will depend on the type of error. In almost all severe error cases, the recovery action will be to terminate the event which has suffered the error. This approach is also required to support minimal human interaction by the Operations staff. This helps support the concept of Production Processing occurring in an unattended mode.

d. Human interaction requirements to support Production Processing must be minimized. To support this, almost all errors will be logged and the Operations staff will be alerted. Depending on whether Production Processing is occurring in an attended operations mode or unattended operations mode, the operations staff will have the capability to manually intervene to correct the error condition. Otherwise, the activity will be terminated by Processing, and a new processing activity will be initiated.

e. Isolation of errors so as not to affect other Processing activities. Science SW processing is terminated to isolate the cause of the error. Also, Processing HW may be taken off-line to correct the resource problems to support the isolation of the error condition.

4.1.1.5 Processing/Planning Interfaces

This section provides a brief description of the relationship between the Planning and Processing CSCIs.

The interface between the Planning and Processing CSCIs occurs through a common shared database, known as the PDPS Database. The PDPS Database is an RDBMS, SYBASE, and contains all data which is persistent to applications associated with Planning, Processing, and Algorithm Integration and Test. The decision to share a common database was driven by the many common data structures which were apparent in the Preliminary Design Specifications developed for the Planning and Data Processing Subsystems.

When a Data Processing Request is planned for execution, as represented in the activated production plan which is created and maintained by Planning CSCI components, the knowledge of this Data Processing Request is transferred into the Processing CSCI domain. This involves adding the definition of a series of jobs representing the Data Processing Request to the COTS product being used to manage production by the Processing CSCI. Please see Section 4.1.3 for more information on the selected COTS product. As these jobs run, the Planning CSCI is capable of polling on the COTS product to retrieve status information for a given job.

4.1.1.6 Processing/MSS Interfaces

This section provides a brief description of the relationship between the Data Processing and MSS Subsystems. A high-level description of the interfaces between MSS and Processing is provided.

The Processing CSCI is dependent on the MSS to provide life cycle services. These services consist of information related to the following activities:

a. Startup of the Processing CSCI software components.

b. Shutdown of the Processing CSCI software components.

MSS provides proxy agents which will be used to communicate startup and shutdown commands to all ECS applications. It is currently envisioned that the MSS proxy agent initiates the PDPS Database. The startup of the PDPS Database will then initiate the startup of the Processing CSCI COTS components. The COTS components will then initiate the custom Processing CSCI components on an as needed basis.

Also, MSS provides mechanisms which enable the Processing CSCI components to provide system management information, such as system-wide event information, resource fault information, accounting and configuration management information**,** and ECS application fault information, to MSS for logging purposes and to initiate system-wide error recovery activities.

In support of system resource configuration management, MSS provides resource configuration information to the Processing CSCI which allows the Processing CSCI to logically manage the allocation of resources to support science data production. MSS will support the monitoring of science software by providing fault isolation and performance tools which provide feedback on the utilization of the science data processing resources.

4.1.1.7 Processing/Data Server Interfaces

The Processing CSCI has an interface to the Science Data Server CSCI to support the staging (Acquiring of data from the Science Data Server CSCI) and destaging (Insertion of data to the Science Data Server CSCI) of data. This interface is a client/server interface with the Processing CSCI acting as the client and the Science Data Server CSCI providing data storage services. The Processing CSCI requests the Science Data Server CSCI to stage or destage data as required to support the generation of a data product. The Processing CSCI will inform the Science Data Server CSCI where the data should be staged (what resource) or where the data should be destaged (copied) from. The Science Data Server CSCI uses an FTP PUSH to copy the data to the science processing resources to support staging of data and uses an FTP PULL to copy the data from the science processing resources to support destaging of data. When the Science Data Server CSCI has completed this task, the Science Data Server CSCI informs the Processing CSCI that the data has been staged or destaged successfully. The Processing CSCI is responsible for determining whether data should be deleted from the science processing resources. When data is destaged, it is considered a copy operation, not a move operation.

4.1.2 Historical Design Transitions

4.1.2.1 Processing CSCI Design Modifications from Release A PDR to Release B IDR

After the Release A Preliminary Design Specification was published, a COTS product(s) was selected which fulfills the majority of the Level 4 requirements. This COTS product has had a tremendous impact on the design as presented here. The following information will summarize these design modifications and their rationale. The selected COTS products are Platinum Technology's AutoSys and AutoXpert. They will be integrated into PDPS to provide the basis for the monitoring and management of ECS' science data production facility.

All design decisions have been driven by a desire to minimize custom code development, tempered by the need to provide proper encapsulation of the COTS to insure later flexibility of adding or modifying the underlying COTS product as ECS matures and evolves.

As a result of these efforts, some design elements which were mapped to the Planning CSCI at the Release A PDR have since moved to the Processing CSCI. This has resulted in a clearer division of the roles and responsibilities of the Planning and Processing CSCIs. As a side effect, the Planning and Processing CSCIs are now more loosely coupled. This will ensure greater flexibility in the future.

The following summarizes the design decisions and provides a top-level view of the current Planning and Data Processing Subsystem Architecture.

1. PDPS will share a common database, i.e., one instance of a SYBASE RDBMS. This will allow PDPS to eliminate the large amount of common persistent data structures which existed in the PDPS preliminary design. For detailed information on the PDPS Database, please refer to Section 4.6.6, PDPS Database CSC, in the Planning Subsystem Preliminary Design Specification
2. The Production Management CSC which was mapped to the Planning CSCI has been divided between the Planning and Processing CSCIs. As presented at the Release A PDR, the Production Management CSC provided two important functions; managing of subscription notifications from the Data Server and Ingest and managing the active plan by receiving status feedback from the Processing CSCI. Since the AutoXpert product provides mechanisms for monitoring and managing the active plan, it was decided to encapsulate the COTS products into a single COTS CSC within the PRONG CSCI. This will provide a more consistent and simpler design with fewer interfaces needed between the Planning and Processing CSCI. Therefore, active plan management is now within the Processing CSCI, whereas, the management of subscription notifications remains in the Planning CSCI.
3. The interface between the Planning and Processing CSCIs has been modified. This change involves when a Data Processing Request is made visible to the Processing CSCI. At the Release A PDR, the approach amounted to not providing a Data Processing Request to the Processing CSCI until all the data subscriptions, sometimes called data dependencies, were fulfilled for a Data Processing Request. Because of the selection of AutoSys and its capabilities to manage job dependencies, this approach has been changed to consist of all "routine" Data Processing Requests being fed into AutoSys at the beginning of the day. The Data Processing Requests which do not have all data dependencies fulfilled would be kept in a "HELD" state until the dependencies are fulfilled. Upon the meeting of all data dependencies, the Planning CSCI would release the Job. Any On-Demand processing requests will be fed directly into AutoSys, where they too will be "HELD" until all dependencies are fulfilled.
4. The software components of the Science Data Pre-Processing CSCI as defined in the Preliminary Design Specification have been mapped into the Processing CSCI or Ingest CSCI, based on what is the optimal location to perform these operations. Within the Processing CSCI, the Science Data Pre-Processing functions have been mapped to a CSC called Data Pre-Processing.

4.1.2.2 Processing CSCI Design Modifications since Release B IDR

Since Rel B IDR , designs have been modified or added in predictive staging, delayed job creation and DAAC automated QA (also called non-Science QA) . The added predictive staging mechanism stages input data granules which are available before a PGE is released. It predicts the appropriate time to start the predictive data staging according to historic data on the PGE staging time and the expected current PGE start time. The delayed job creation design moves all the job creations except the job box and the resource allocation to the execution of preparation job. This design helps to reduce the number of jobs that AutoSys has to handle at any give time. DAAC automated QA allows a check of the production process used to generated an output product.

4.1.3 COTS Strategy

This COTS Strategy section summarizes the objectives and technical approach which was taken to determine the optimal COTS solution required to support the job scheduling functions associated with the Processing CSCI management of the science processing resources. There were four objectives which influenced the decision on the type of COTS products selected;

1) Minimize custom code development

   To insure an adequate solution for the Release A time frame, the COTS solution had to minimize the amount of custom code development which was required to provide the complete Planning and Data Processing Subsystem software solution. This was necessary to mitigate the schedule risk associated with Release A which because of time constraints, would not support a large growth in custom code development. Carrying the COTS solution over to Release B (where ever possible) further minimizes code development and relieves schedule pressure on Release B development.

2) Reach COTS decision to support Release A Detailed Design and Implementation Phase.

   An early decision on the COTS product was required to insure that the required modifications to the Planning and Processing CSCI design could be made to support Release A CDR and to ultimately support the Release A software development schedule. Carrying this solution over to Release B insures that new functionality design can be made to support Release B IDR, CDR, and the eventual code development schedules.

3) Ease of integration into ECS software applications.

   The selected COTS product must support command line or API style interfaces to support the extensive integration efforts which must occur to meet ECS requirements.

4) Scalability to Release B processing load.

   The selected COTS product must be capable of supporting large numbers of jobs. This is necessary to support the Release B processing load.

The adopted technical approach consisted of the following steps:

1) Collecting information about the different types of COTS which could be used to support ECS Planning and Processing functions;
   a) Vendor teleconferences and meetings
   b) Customer teleconferences
   c) Vendor site visits
2) Determining different types of software architecture given the COTS products and the unique ECS Planning and Processing requirements.

3) Analyzing the different classes of COTS packages to determine viability in ECS given the previously defined objectives.

The information gained in this process was used during the preparation of the RFP (Request For Proposal). This RFP was divided into mandatory, optional, and implementation features sections. The responses to these proposals were analyzed and scored based on the information provided by the vendor for each of these sections.

4.1.4 COTS Selection

The following sections provide information on the COTS products selected to support the Processing CSCI in performing management and monitoring activities associated with ECS' science data production environment. The product selected was Platinum Technology's AutoSys and AutoXpert products. A summary of the AutoSys' capabilities and a scenario which explains the set of actions taken to initiate a job is provided.

4.1.4.1 Platinum Technology's AutoSys

AutoSys is a job scheduling software application used to automate operations in a distributed UNIX environment. AutoSys performs automated job control functions required for scheduling, monitoring, and reporting on jobs that reside on any machine attached to a network. In ECS, the machines for which AutoSys will be responsible are the Science Processing hardware resources. In AutoSys, a job is defined as any UNIX command or shell script plus a variety of qualifying attributes which include conditions specifying when and where the job should be run.

AutoSys provides a complete system solution to support job scheduling. This includes an Operator Console, which allows human intervention into the AutoSys job stream, and provides various mechanisms for monitoring the AutoSys job stream. Provided with the interface are capabilities to view job definitions, change the state of a job, modify the job definition, and alter job dependency information. Also provided with the Operations interface is an alarm manager. The alarm manager can be used to assist in monitoring jobs and to react to fault situations. These alarms can be set through the definition of a job. More details on the underlying components of AutoSys software are contained in Section 4.6, CSCI Structure. More information about the AutoSys Operator Console is contained in Section 4.7.2, Operator Interfaces.

4.1.4.2 AutoSys Integration into the Processing CSCI Design

For the Processing CSCI, AutoSys provides the job scheduling engine for the Processing CSCI. AutoSys' Event Processor will manage all the events that occur in the science data production environment. AutoSys' Database will become part of the PDPS Database using the AutoSys provided table schema. For detailed design, the current assumption is that the AutoSys provided processes will manage the processing environment when the production facility is operating in a steady state manor. For start up and shutdown, some development code is needed to assist in establishing communication connections, initializing the PDPS database, assuring the availability of other entities, i.e., Data Server, and alerting operations and other applications about startup and initialization problems. After the completion of startup and upon the completion of adding the daily job schedule to AutoSys, AutoSys will begin managing and monitoring the execution of jobs.

To support the execution of jobs, AutoSys will require additional help in the following areas:

a. **Resource Management—**Allocation of sufficient resources, i.e., disk storage, to support execution. Currently, AutoSys provides no mechanisms for managing disk storage. This is a potential enhancement that will be added to their product. Also, the monitoring of resources will not be done by AutoSys. This is a MSS and Processing joint responsibility.

b. **Data Management—**Manages the staging, destaging, and retention of data on Processing resources.

c. **PGE Execution Management—**Initiates and monitors execution of a PGE.

The following paragraphs briefly illustrates the operational concepts of the Processing CSCI design. They discuss the following:

a. How a job schedule is created

b. How jobs are prepared and initiated, and

c. How post-processing takes place.

These applications will be initiated by AutoSys at the Job level. For each PGE, a series of preparation, execution, and post-execution jobs will be defined. The preparation jobs will manage the staging of data (if necessary) and allocation of resources to support execution. The execution job will be used to initiate and monitor the execution of the PGE. The post-execution job will be used to destage and deallocate resources.

To accomplish the set-up of these jobs, AutoSys' capabilities to create and manage job dependencies and to create job boxes, which consist of a series of related jobs, will be used. For each Data Processing Request, which contains the data required to support the execution of one PGE, received from Planning, a corresponding job box, which contains a series of jobs to allocate resources, stage data, execute the PGE, destage data, and deallocate resources, will be created. In Figure 4.1-1, the diagram shows the steps involved in providing job information to AutoSys. The Production Planning Workbench component of the Planning CSCI is the initiator of this activity. The Production Planning Workbench component will use the Scheduler (DpPrScheduler) public class which is being created to provide an interface to AutoSys. This class will encapsulate AutoSys defined capabilities and will manage the information flow from Planning to AutoSys. Through the use of API and command line interfaces, the Scheduler class will provide job definitions to the AutoSys Database. Also, through the DpPrScheduler, the Planning CSCI will be able to request the status for the jobs associated with a Data Processing Request.

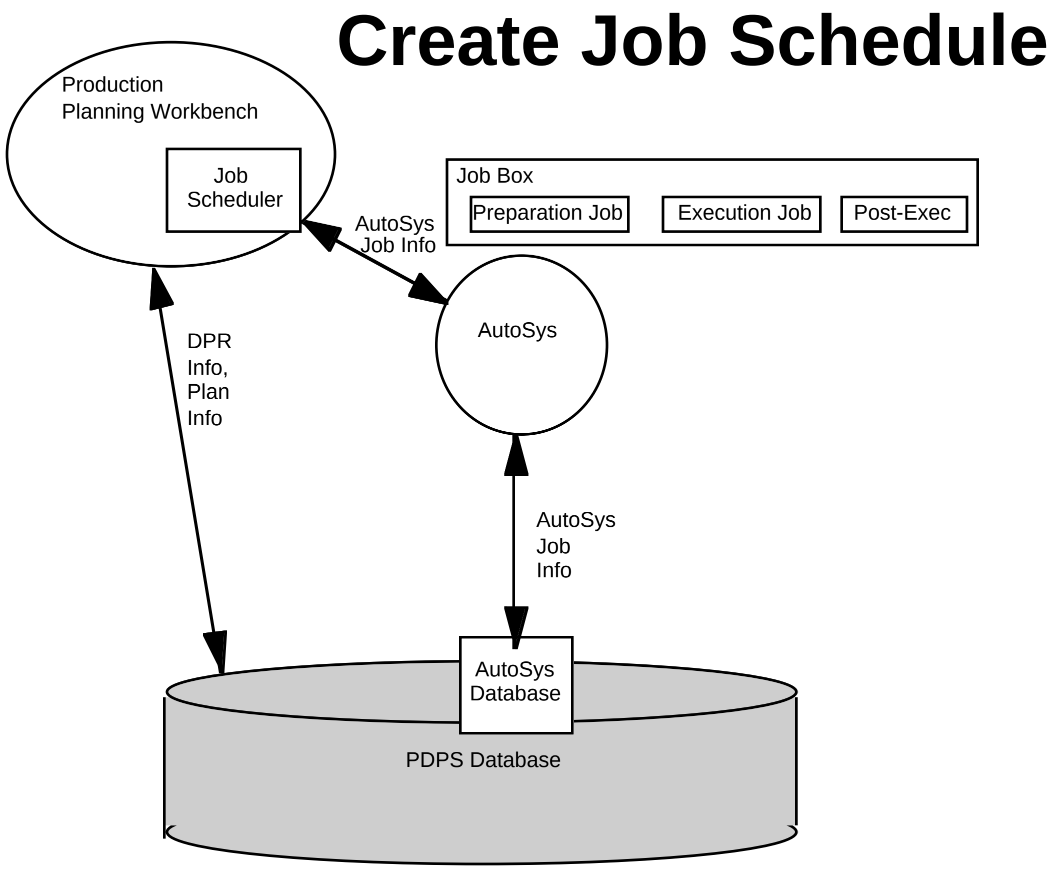


Figure 4.1-1. Scheduling Jobs Using AutoSys

Each of these jobs will actually consist of a command which initiates a process or processes to perform the task desired. The process will perform the desired functions and gracefully terminate. The successful completion of their task will result in the next dependent job being released until all jobs within the job box have completed. Besides managing normal operations through these mechanisms, failures which occur within a job box could result in the execution of special fault recovery jobs when necessary.

The Processing CSCI custom components will be initiated by AutoSys to perform support activities, such as resource allocation, staging and destaging data, and interfacing with the PGE. Figure 4.1-2 shows the series of steps and resulting actions performed by the Processing CSCI components.

Initiate Preparation Job

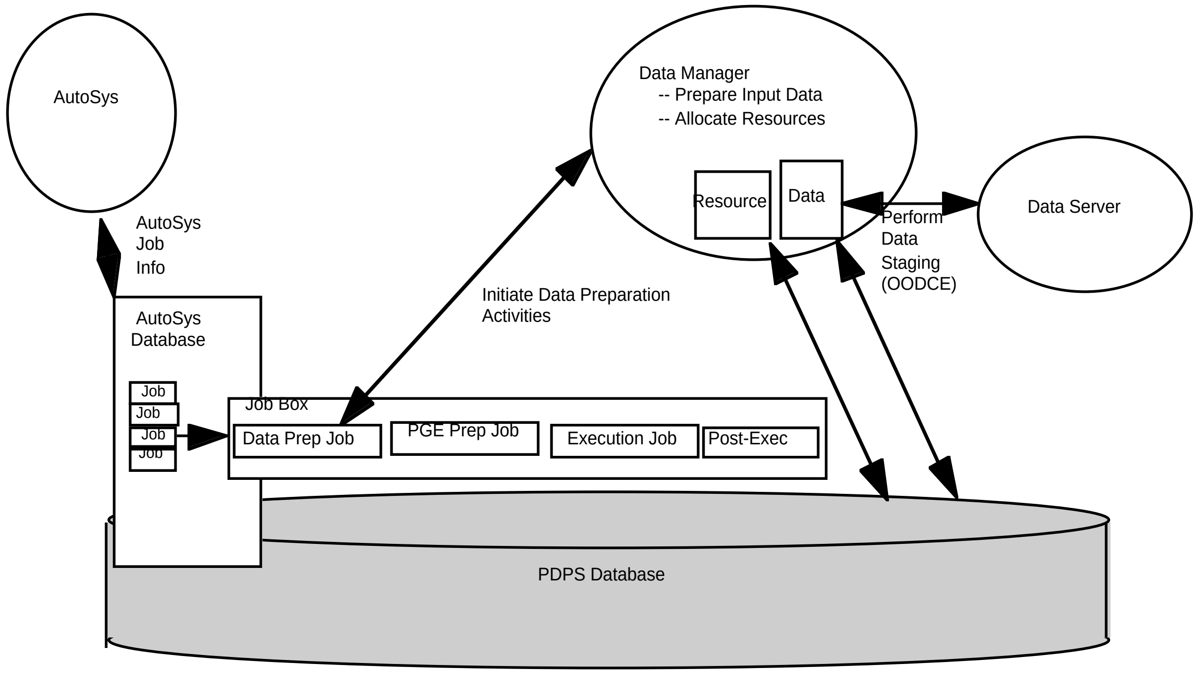


Figure 4.1-2. Initiating Processing Components Using AutoSys

AutoSys also provides logging and report generation in order to capture information on job results and status changes. This information will provide information previously mapped to the Processing Log (DpPrProcessingLog) as presented in the Preliminary Design Specification.

4.1.4.3 Platinum Technology's AutoXpert

This product provides mechanisms to monitor and manage the job schedule which currently is being processed in AutoSys. This product is a GUI which provides different methods of viewing the progress of the job schedule, determining corrective measures when required, and providing capabilities to play what-if simulations to determine the affects of modifying the active job schedule. AutoXpert allows the job schedule to be represented at many different levels of abstraction. This concept is an important concept given the large numbers of jobs expected to executed per day at a given DAAC site. These abstract views include a time-line, gantt chart, and job data flows. As part of these views, jobs which are not following predicted behaviors will be color coded which will allow the operations staff to intervene, if necessary. More information about the AutoSys Operator Console and the AutoXpert GUI capabilities is contained in Section 4.6.2, Operator Interfaces.

4.1.5 Database Interface

A group of PDPS database interface classes will provide an efficient mechanism for interfacing with the database. This interface provides the following features:

- encapsulates most of the Rogue Wave DBtools.h++ classes so that application developers do not have to be familiar with DBtools protocol in order to access the database
- manages centralized database connections within an application so that objects do not make unnecessary database connections
- allows developers to easily manipulate data without having to translate between class attribute names and database table column names

4.2 CSCI Context

The Processing CSCI interfaces with the following external actors, SDPS and CSMS subsystems to fulfill its responsibilities:

a. Planning Subsystem—The Planning Subsystem is responsible for creating a production plan for the Processing CSCI. The Production plan information is conveyed to the Processing CSCI through the use of Data Processing Requests. Each Data Processing Request represents one processing job to be performed by a Data Processing Subsystem computer resource. Processing will provide status information to Planning to assist in production management activities of the Planning CSCI.

b. Data Server Subsystem—To support the creation of ECS Data Products, the Processing CSCI needs the capability of requesting and receiving data (Data Staging) from any of the Data Server resources which has the responsibility of maintaining the data. Also, the Processing CSCI needs the capability of transferring data (Data Destaging) to any Data Server resource for archiving of a generated data product.

c. Operations Interface—To support the management and monitoring of the execution of a PGE and the creation of ECS Data Products, a HMI interface is provided. This interface will provide services to support the collection of status for a Data Processing Request, the cancellation, suspension/resumption and/or modification of a Data Processing Request, and monitoring of the health of Data Processing Subsystem hardware resources. Also, this interface will be used to support manual quality assurance operations performed at the DAAC.

d. SDP Toolkit Interface—To support PGE execution, the Processing CSCI provides information on the location of input data and the location of where the generated output data products are to be maintained. While a PGE is executing, the Processing CSCI monitors the execution of the PGE and informs the operations staff of current status. Status may include current processing event history (what is happening, i.e., data staging, execution). Also, monitoring will be needed to make sure that the processing activity is executing properly. Upon completion of the execution of a PGE, the Data Processing CSCIwill inform Planning and will initiate the transfer of the generated data product (if necessary) to the Data Server.

e. CSMS Interface(s)—The Processing CSCI relies on CSMS services to assist in communications and resource management activities and provides system management information to CSMS for Fault, Accounting, Configuration, Security, Performance, and Accountability purposes. Also, CSMS will provide support for the monitoring of Processing resources and the execution of Science Algorithms.

f. Ingest CSCI—To support science data production, the Processing CSCI must be capable of acquiring necessary input data, the Ingest CSCI's hardware resources are the location of the Level 0 data, and other additional products when first input into ECS.

4.3 CSCI Object Model

The Processing CSCI Object Model is actually composed of a number of differing views of components which compose the Processing CSCI. These components provide different abstract views of the Processing CSCI design to assist in developing an understanding of the design. The following object model views will be presented and the underlying objects will be described:

1. Processing CSCI top-level CSC view
2. COTS Manager
3. PGE Execution Manager
4. Data Manager
5. Resource Manager
6. Q/A Monitor
7. DAAC Non-Science QA
8. Data Pre-Processing

Besides the above listed Processing CSCI components, the Processing CSCI is dependent on the PDPS Database for the management of Processing CSCI persistent data storage. More information on the PDPS Database is contained in the PDPS Database CSC of the Planning Subsystem Design Specification.

4.3.1 Processing CSCI Component View

This view of the Processing CSCI (Figure 4.3-1) represents the various components and the associations of these components of the Processing CSCI.

AutoXpert provides additional capabilities to perform simulations using the current active job schedule. These capabilities will assist the operations staff in judging the downstream effects of the failure of a job, the unavailability of a resource, the late arrival of data, and a job running longer than predicted. These capabilities are provided and can be used in real-time for projections, but also in a simulation mode, where the schedule can be sped up or other job information can be changed to judge what the impact would be. These additional capabilities will help assist DAAC operations personnel in determining what impact production anomalies will have against the active plan and may be used to determine if a replan of production is necessary.

In terms of the Release A PDPS Preliminary Design Specification, AutoXpert provides functions which were previously mapped to the Production Management CSC in the Planning CSCI. Since these functions are being provided by the AutoSys and AutoXpert products, it seemed desirable to map the COTS product to one CSCI. This decision eliminated the need to represent interfaces between AutoSys and AutoXpert across CSCI and Subsystem boundaries.

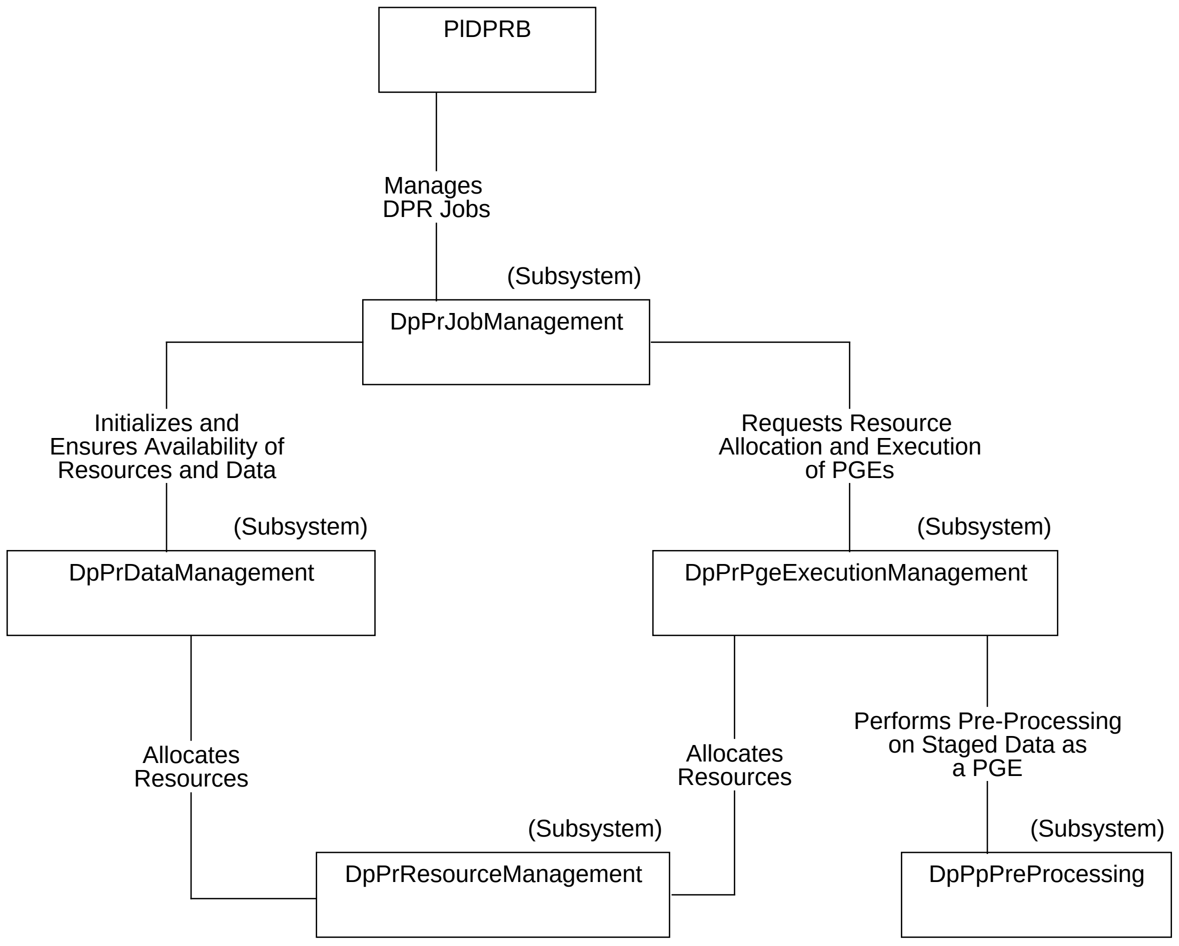


Figure 4.3-1. Processing CSC Level Object Model

4.3.2 COTS Manager View

The COTS Manager View (Figure 4.3-2) represents the classes used to manage the flow of information to the COTS products. All ECS applications which need to pass information to the COTS products, which include Planning CSCI components and other Processing CSCI components, will interface to the COTS using these classes. This collection of classes provides the interface to AutoSys, and will take information provided and translate to the COTS supplied API and command line interfaces. Any application which requires an interface to AutoSys would be required to use this interface. The following functions are provided through the COTS Manager:

1. Manages the start-up and shut-down of the AutoSys and AutoXpert products.
2. Provides the interface to AutoSys to add, modify, suspend, resume, and cancel jobs (Planning CSCI interfaces).
3. Provides predictive staging scheme to achieve efficient resource utilization.
4. Provides additional management/monitoring mechanisms (if needed)
5. Provides the interface to allow the MSS CSCI to interact with AutoSys to alert AutoSys of unavailable resources and external event information, and for AutoSys to provide information to MSS (including, accounting, configuration, Report/log information, if necessary).

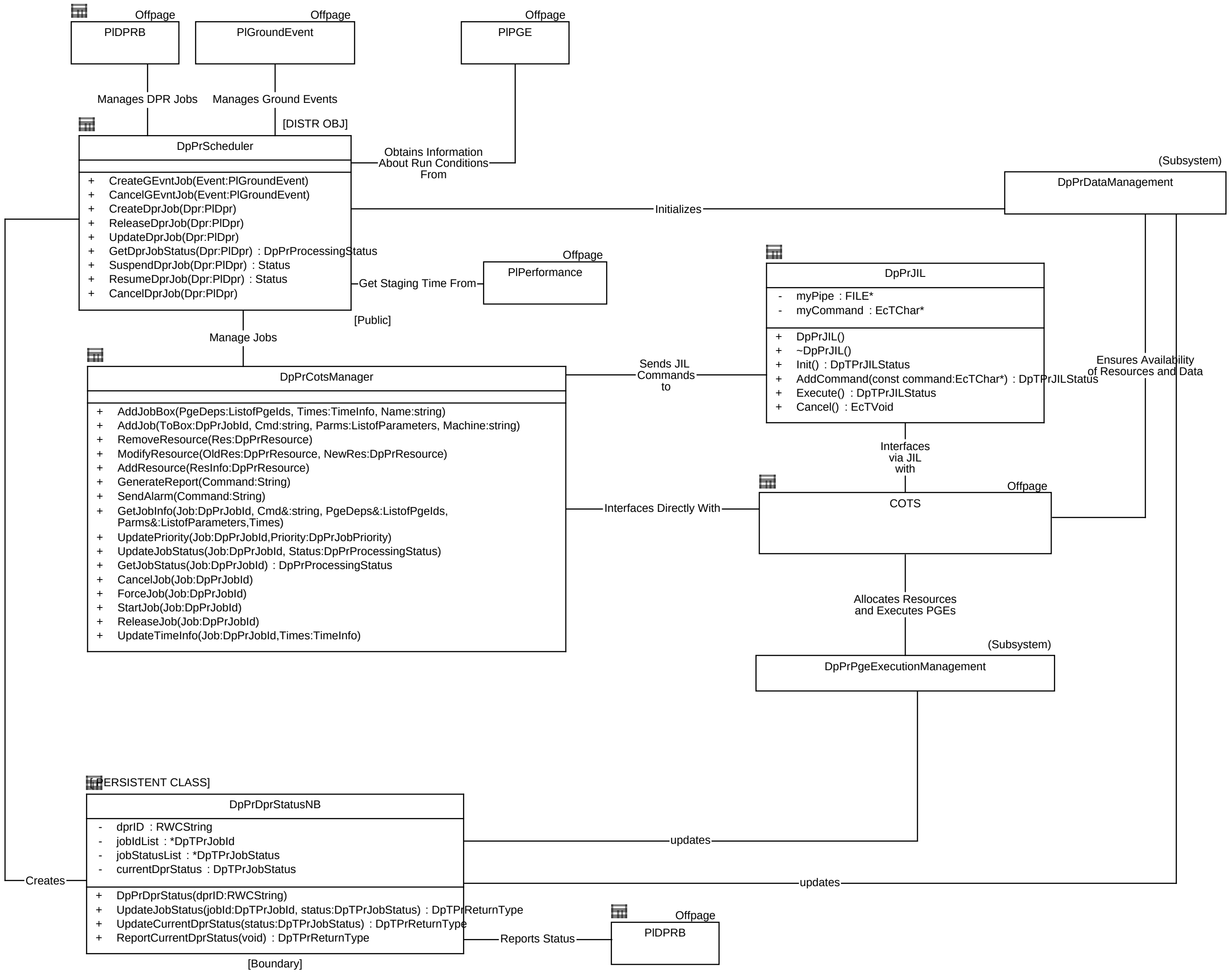

Offpage
PlDPRB
Offpage
PlGroundEvent
Offpage
PlPGE
Manages DPR Jobs
Manages Ground Events
[DISTR OBJ]
DpPrScheduler
+ CreateGEvntJob(Event:PlGroundEvent)
+ CancelGEvntJob(Event:PlGroundEvent)
+ CreateDprJob(Dpr:PlDpr)
+ ReleaseDprJob(Dpr:PlDpr)
+ UpdateDprJob(Dpr:PlDpr)
+ GetDprJobStatus(Dpr:PlDpr) : DpPrProcessingStatus
+ SuspendDprJob(Dpr:PlDpr) : Status
+ ResumeDprJob(Dpr:PlDpr) : Status
+ CancelDprJob(Dpr:PlDpr)
Obtains Information About Run Conditions From
(Subsystem)
DpPrDataManagement
Initializes
Offpage
PlPerformance
Get Staging Time From
[Public]
Manage Jobs
DpPrJIL
- myPipe : FILE*
- myCommand : EcTChar*
+ DpPrJIL()
+ ~DpPrJIL()
+ Init() : DpTPrJILStatus
+ AddCommand(const command:EcTChar*) : DpTPrJILStatus
+ Execute() : DpTPrJILStatus
+ Cancel() : EcTVoid
Ensures Availability of Resources and Data
DpPrCotsManager
+ AddJobBox(PgeDeps:ListofPgeIds, Times:TimeInfo, Name:string)
+ AddJob(ToBox:DpPrJobId, Cmd:string, Parms:ListofParameters, Machine:string)
+ RemoveResource(Res:DpPrResource)
+ ModifyResource(OldRes:DpPrResource, NewRes:DpPrResource)
+ AddResource(ResInfo:DpPrResource)
+ GenerateReport(Command:String)
+ SendAlarm(Command:String)
+ GetJobInfo(Job:DpPrJobId, Cmd&:string, PgeDeps&:ListofPgeIds, Parms&:ListofParameters,Times)
+ UpdatePriority(Job:DpPrJobId,Priority:DpPrJobPriority)
+ UpdateJobStatus(Job:DpPrJobId, Status:DpPrProcessingStatus)
+ GetJobStatus(Job:DpPrJobId) : DpPrProcessingStatus
+ CancelJob(Job:DpPrJobId)
+ ForceJob(Job:DpPrJobId)
+ StartJob(Job:DpPrJobId)
+ ReleaseJob(Job:DpPrJobId)
+ UpdateTimeInfo(Job:DpPrJobId,Times:TimeInfo)
Sends JIL Commands to
Interfaces via JIL with
Offpage
COTS
Interfaces Directly With
Allocates Resources and Executes PGEs
(Subsystem)
DpPrPgeExecutionManagement
[PERSISTENT CLASS]
DpPrDprStatusNB
- dprID : RWCString
- jobIdList : *DpTPrJobId
- jobStatusList : *DpTPrJobStatus
- currentDprStatus : DpTPrJobStatus
+ DpPrDprStatus(dprID:RWCString)
+ UpdateJobStatus(jobId:DpTPrJobId, status:DpTPrJobStatus) : DpTPrReturnType
+ UpdateCurrentDprStatus(status:DpTPrJobStatus) : DpTPrReturnType
+ ReportCurrentDprStatus(void) : DpTPrReturnType
[Boundary]
Creates
updates
updates
Reports Status
Offpage
PlDPRB

Figure 4.3-2. COTS Manager Object Model

4.3.3 Data Management View

The Data Management View (Figure 4.3-3) represents the classes used to manage the flow of science data to and from science processing resources. This includes the communication mechanisms needed to interface with the Science Data Server CSCI and the Ingest CSCI. Also, these classes provide additional functions to manage the retention of data on science processing resources which is used to support many PGE executions.

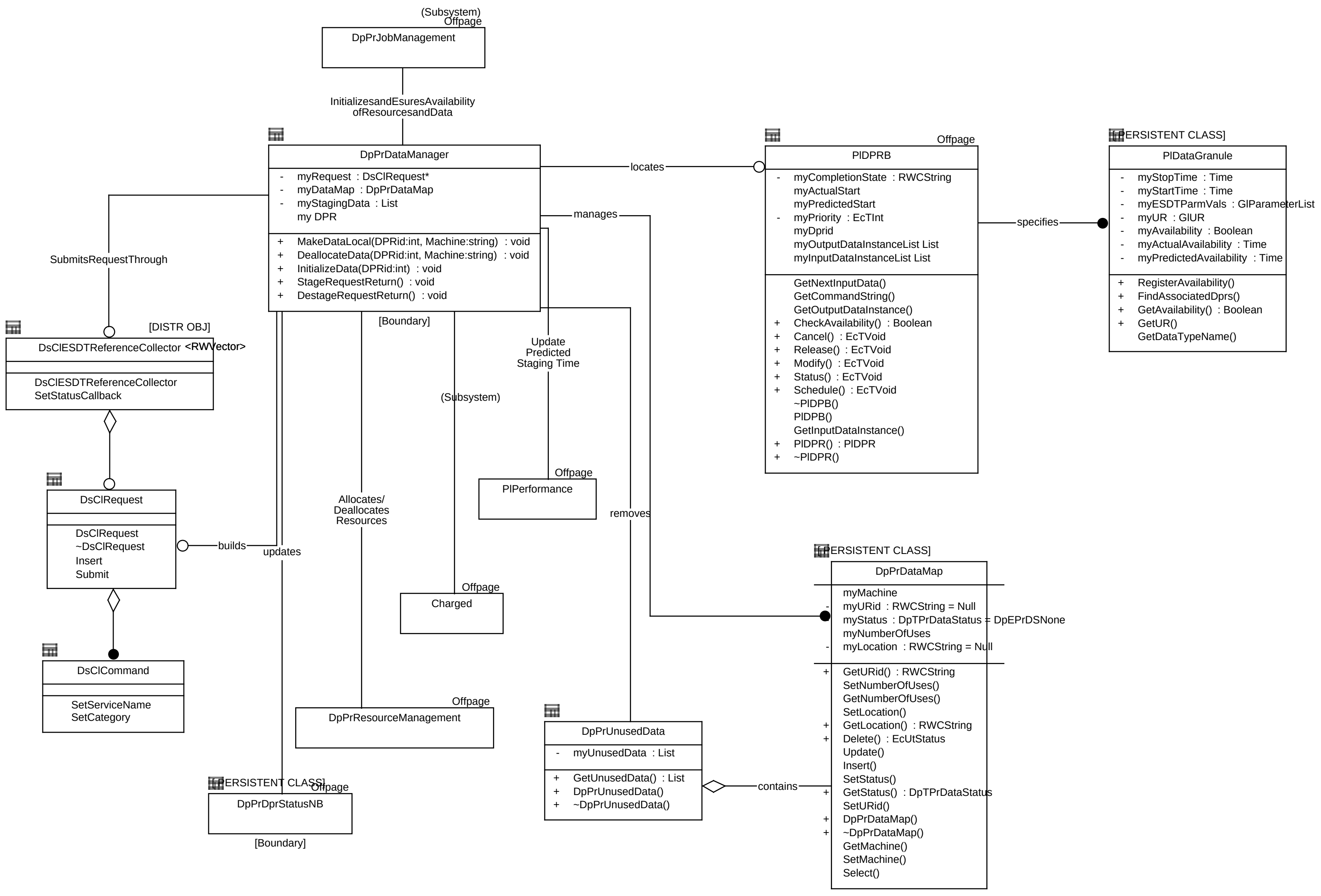


Figure 4.3-3. Data Management Object Model

4.3.4 PGE Execution Management View

The PGE Execution Manager View (Figure 4.3-4) represents the classes used to support the execution of a PGE. While the COTS product will actually initiate the execution of the PGE, the supporting preparation activities, such as creating the Process Control File, are provided through these classes.

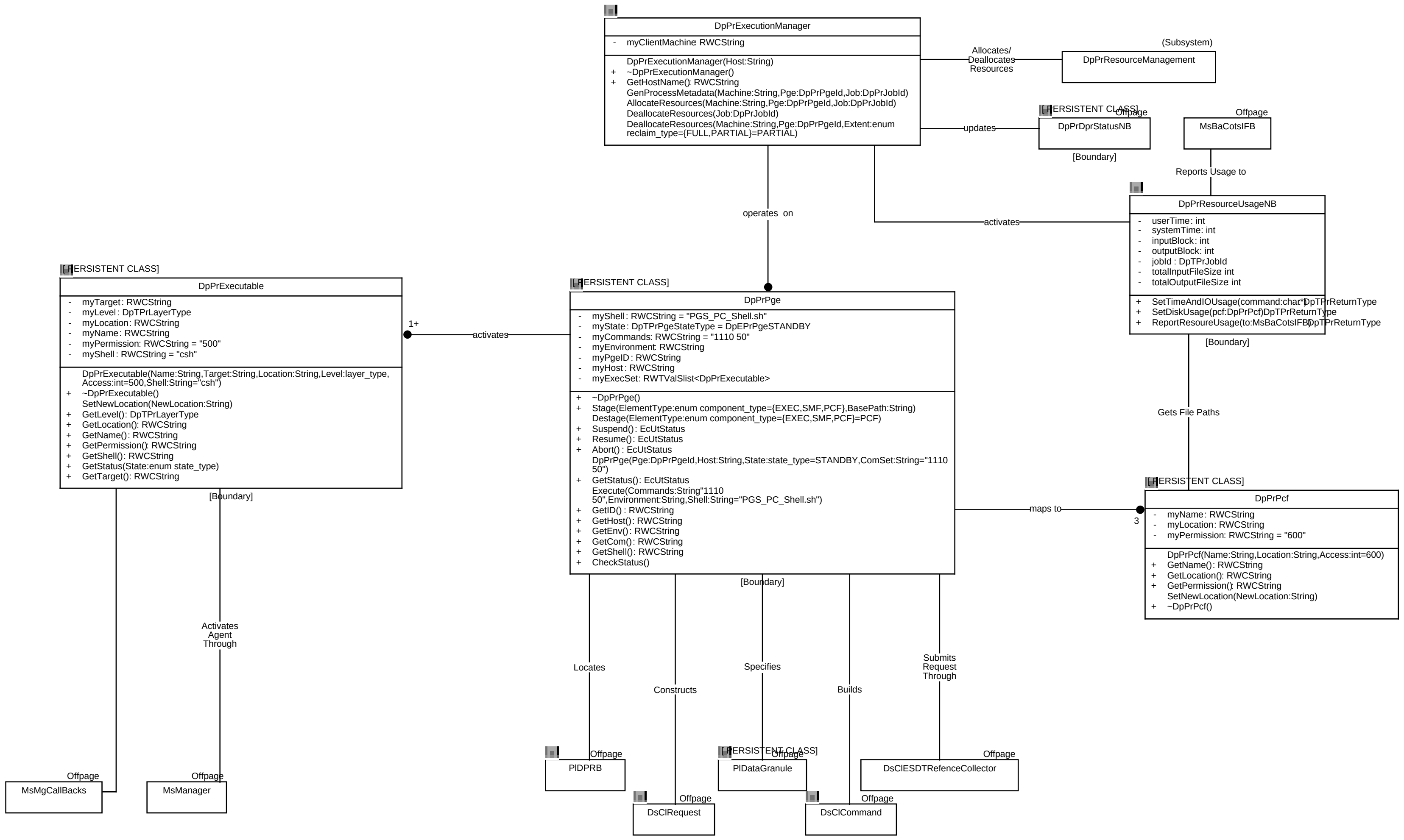
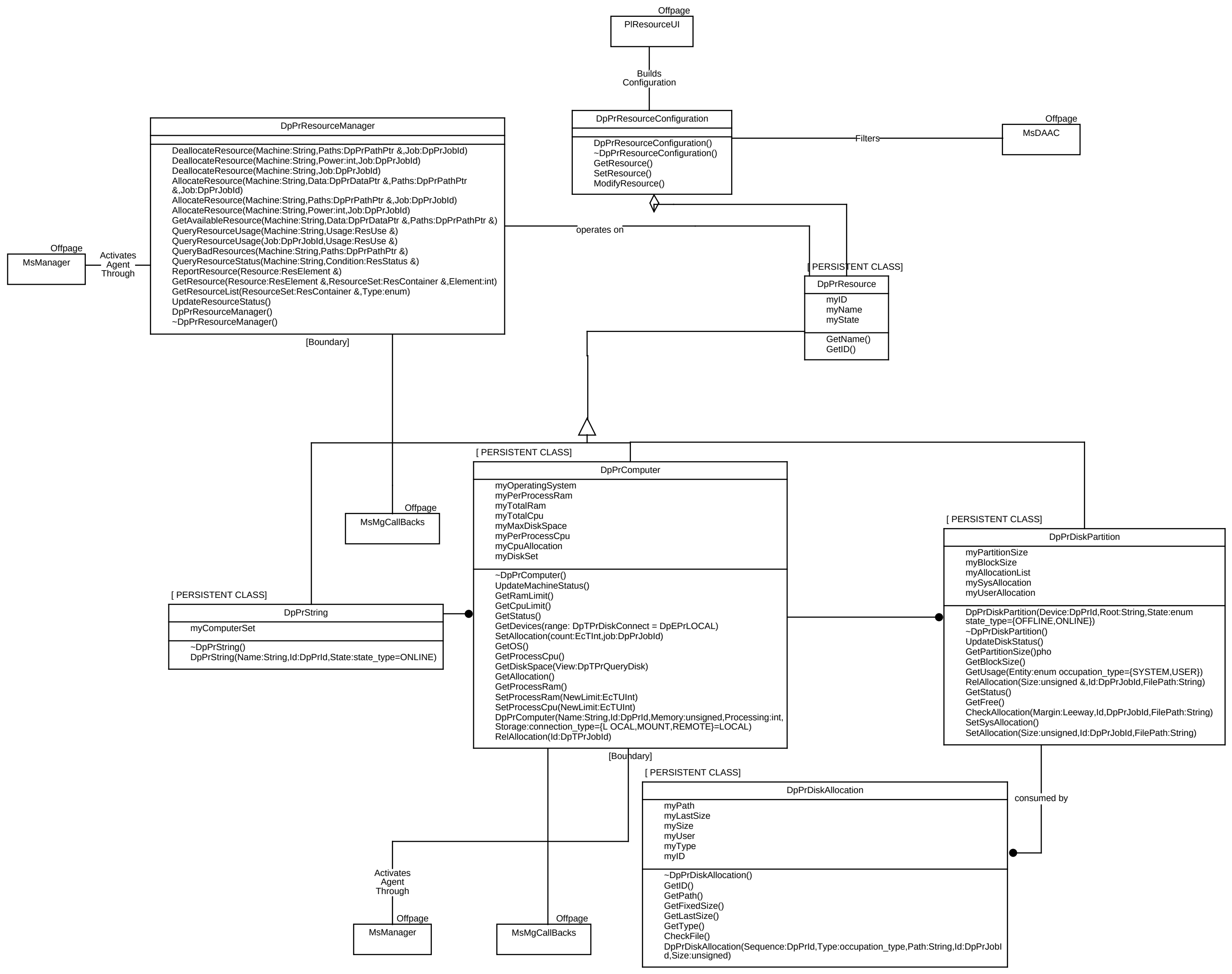

DpPrExecutionManager
- myClientMachine: RWCString
DpPrExecutionManager(Host:String)
+ ~DpPrExecutionManager()
+ GetHostName(): RWCString
GenProcessMetadata(Machine:String,Pge:DpPrPgeId,Job:DpPrJobId)
AllocateResources(Machine:String,Pge:DpPrPgeId,Job:DpPrJobId)
DeallocateResources(Job:DpPrJobId)
DeallocateResources(Machine:String,Pge:DpPrPgeId,Extent:enum reclaim_type={FULL,PARTIAL}=PARTIAL)
Allocates/ Deallocates Resources
(Subsystem)
DpPrResourceManagement
updates
[PERSISTENT CLASS]
Offpage
DpPrDprStatusNB
[Boundary]
Offpage
MsBaCotsIFB
Reports Usage to
activates
operates on
DpPrResourceUsageNB
- userTime: int
- systemTime: int
- inputBlock: int
- outputBlock: int
- jobId : DpTPrJobId
- totalInputFileSize int
- totalOutputFileSize int
+ SetTimeAndIOUsage(command:char*):DpTPrReturnType
+ SetDiskUsage(pcf:DpPrPcf):DpTPrReturnType
+ ReportResoureUsage(to:MsBaCotsIFB):DpTPrReturnType
[Boundary]
Gets File Paths
[PERSISTENT CLASS]
DpPrExecutable
- myTarget: RWCString
- myLevel: DpTPrLayerType
- myLocation: RWCString
- myName: RWCString
- myPermission: RWCString = "500"
- myShell: RWCString = "csh"
DpPrExecutable(Name:String,Target:String,Location:String,Level:layer_type, Access:int=500,Shell:String="csh")
+ ~DpPrExecutable()
SetNewLocation(NewLocation:String)
+ GetLevel(): DpTPrLayerType
+ GetLocation(): RWCString
+ GetName(): RWCString
+ GetPermission(): RWCString
+ GetShell(): RWCString
+ GetStatus(State:enum state_type)
+ GetTarget(): RWCString
[Boundary]
1+
activates
[PERSISTENT CLASS]
DpPrPge
- myShell: RWCString = "PGS_PC_Shell.sh"
- myState: DpTPrPgeStateType = DpEPrPgeSTANDBY
- myCommands: RWCString = "1110 50"
- myEnvironment: RWCString
- myPgeID: RWCString
- myHost: RWCString
- myExecSet: RWTValSlist<DpPrExecutable>
+ ~DpPrPge()
+ Stage(ElementType:enum component_type={EXEC,SMF,PCF},BasePath:String)
Destage(ElementType:enum component_type={EXEC,SMF,PCF}=PCF)
+ Suspend(): EcUtStatus
+ Resume(): EcUtStatus
+ Abort(): EcUtStatus
DpPrPge(Pge:DpPrPgeId,Host:String,State:state_type=STANDBY,ComSet:String="1110 50")
+ GetStatus(): EcUtStatus
Execute(Commands:String"1110 50",Environment:String,Shell:String="PGS_PC_Shell.sh")
+ GetID(): RWCString
+ GetHost(): RWCString
+ GetEnv(): RWCString
+ GetCom(): RWCString
+ GetShell(): RWCString
+ CheckStatus()
[Boundary]
maps to
3
[PERSISTENT CLASS]
DpPrPcf
- myName: RWCString
- myLocation: RWCString
- myPermission: RWCString = "600"
DpPrPcf(Name:String,Location:String,Access:int=600)
+ GetName(): RWCString
+ GetLocation(): RWCString
+ GetPermission(): RWCString
SetNewLocation(NewLocation:String)
+ ~DpPrPcf()
Activates Agent Through
Locates
Constructs
Specifies
Builds
Submits Request Through
Offpage
MsMgCallBacks
Offpage
MsManager
Offpage
PlDPRB
Offpage
DsClRequest
[PERSISTENT CLASS]
Offpage
PlDataGranule
Offpage
DsClCommand
Offpage
DsClESDTRefenceCollector

Figure 4.3-4. Execution Management Object Model

4.3.5 Resource Management View

The Resource Management View (Figure 4.3-5) represents the classes used to support the management of science processing resources. These classes provide mappings of logical to the physical resources to allow the Processing CSCI to manage and monitor science processing resources being used to support science data production. This process provides additional resource management and monitoring capabilities that are not currently provided by AutoSys. At this time, it is thought that the interface to Resource Management is through AutoSys, i.e., AutoSys will signal or initiate Resource Management when required. These additional functions will help determine whether all required resources are available to support the execution of the PGE. Please note, that this is one area where Platinum Technology has agreed to strengthen AutoSys' capabilities using PDPS input to determine these additional capabilities.



Offpage
PlResourceUI
Builds Configuration
DpPrResourceConfiguration
DpPrResourceConfiguration()
~DpPrResourceConfiguration()
GetResource()
SetResource()
ModifyResource()
Filters
Offpage
MsDAAC
DpPrResourceManager
DeallocateResource(Machine:String,Paths:DpPrPathPtr &,Job:DpPrJobId)
DeallocateResource(Machine:String,Power:int,Job:DpPrJobId)
DeallocateResource(Machine:String,Job:DpPrJobId)
AllocateResource(Machine:String,Data:DpPrDataPtr &,Paths:DpPrPathPtr &,Job:DpPrJobId)
AllocateResource(Machine:String,Paths:DpPrPathPtr &,Job:DpPrJobId)
AllocateResource(Machine:String,Power:int,Job:DpPrJobId)
GetAvailableResource(Machine:String,Data:DpPrDataPtr &,Paths:DpPrPathPtr &)
QueryResourceUsage(Machine:String,Usage:ResUse &)
QueryResourceUsage(Job:DpPrJobId,Usage:ResUse &)
QueryBadResources(Machine:String,Paths:DpPrPathPtr &)
QueryResourceStatus(Machine:String,Condition:ResStatus &)
ReportResource(Resource:ResElement &)
GetResource(Resource:ResElement &,ResourceSet:ResContainer &,Element:int)
GetResourceList(ResourceSet:ResContainer &,Type:enum)
UpdateResourceStatus()
DpPrResourceManager()
~DpPrResourceManager()
[Boundary]
Offpage
MsManager
Activates Agent Through
operates on
[PERSISTENT CLASS]
DpPrResource
myID
myName
myState
GetName()
GetID()
[PERSISTENT CLASS]
DpPrComputer
myOperatingSystem
myPerProcessRam
myTotalRam
myTotalCpu
myMaxDiskSpace
myPerProcessCpu
myCpuAllocation
myDiskSet
~DpPrComputer()
UpdateMachineStatus()
GetRamLimit()
GetCpuLimit()
GetStatus()
GetDevices(range: DpTPrDiskConnect = DpEPrLOCAL)
SetAllocation(count:EcTInt,job:DpPrJobId)
GetOS()
GetProcessCpu()
GetDiskSpace(View:DpTPrQueryDisk)
GetAllocation()
GetProcessRam()
SetProcessRam(NewLimit:EcTUInt)
SetProcessCpu(NewLimit:EcTUInt)
DpPrComputer(Name:String,Id:DpPrId,Memory:unsigned,Processing:int, Storage:connection_type={L OCAL,MOUNT,REMOTE}=LOCAL)
RelAllocation(Id:DpTPrJobId)
[Boundary]
Offpage
MsMgCallBacks
[PERSISTENT CLASS]
DpPrString
myComputerSet
~DpPrString()
DpPrString(Name:String,Id:DpPrId,State:state_type=ONLINE)
[PERSISTENT CLASS]
DpPrDiskPartition
myPartitionSize
myBlockSize
myAllocationList
mySysAllocation
myUserAllocation
DpPrDiskPartition(Device:DpPrId,Root:String,State:enum state_type={OFFLINE,ONLINE})
~DpPrDiskPartition()
UpdateDiskStatus()
GetPartitionSize()pho
GetBlockSize()
GetUsage(Entity:enum occupation_type={SYSTEM,USER})
RelAllocation(Size:unsigned &,Id:DpPrJobId,FilePath:String)
GetStatus()
GetFree()
CheckAllocation(Margin:Leeway,Id,DpPrJobId,FilePath:String)
SetSysAllocation()
SetAllocation(Size:unsigned,Id:DpPrJobId,FilePath:String)
consumed by
[PERSISTENT CLASS]
DpPrDiskAllocation
myPath
myLastSize
mySize
myUser
myType
myID
~DpPrDiskAllocation()
GetID()
GetPath()
GetFixedSize()
GetLastSize()
GetType()
CheckFile()
DpPrDiskAllocation(Sequence:DpPrId,Type:occupation_type,Path:String,Id:DpPrJobId,Size:unsigned)
Activates Agent Through
Offpage
MsManager
Offpage
MsMgCallBacks

Figure 4.3-5. Resource Management Object Model

4.3.6 Quality Assurance Monitor View

The Quality Assurance View (Figure 4.3-6) represents the classes used to support the DAAC operations position used to perform DAAC manual quality assurance activities. These activities include visualization of science data products and updating quality assurance metadata.

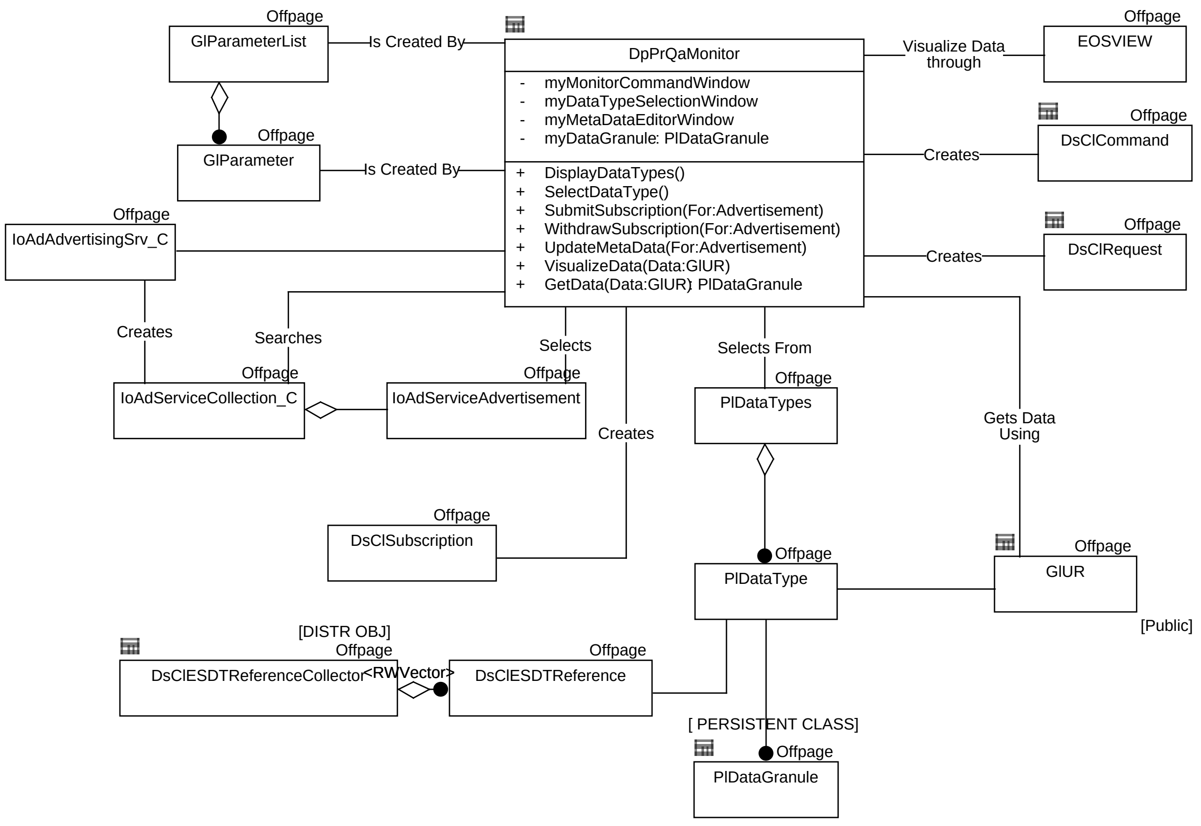


Figure 4.3-6. QA Monitor Object Model

4.3.7 DAAC Non-Science Quality Assurance View

The DAAC Non-Science Quality Assurance View (Figure 4.3-7) represents the classes used to support the automated quality assurance processing which collects and reports information about the production process.

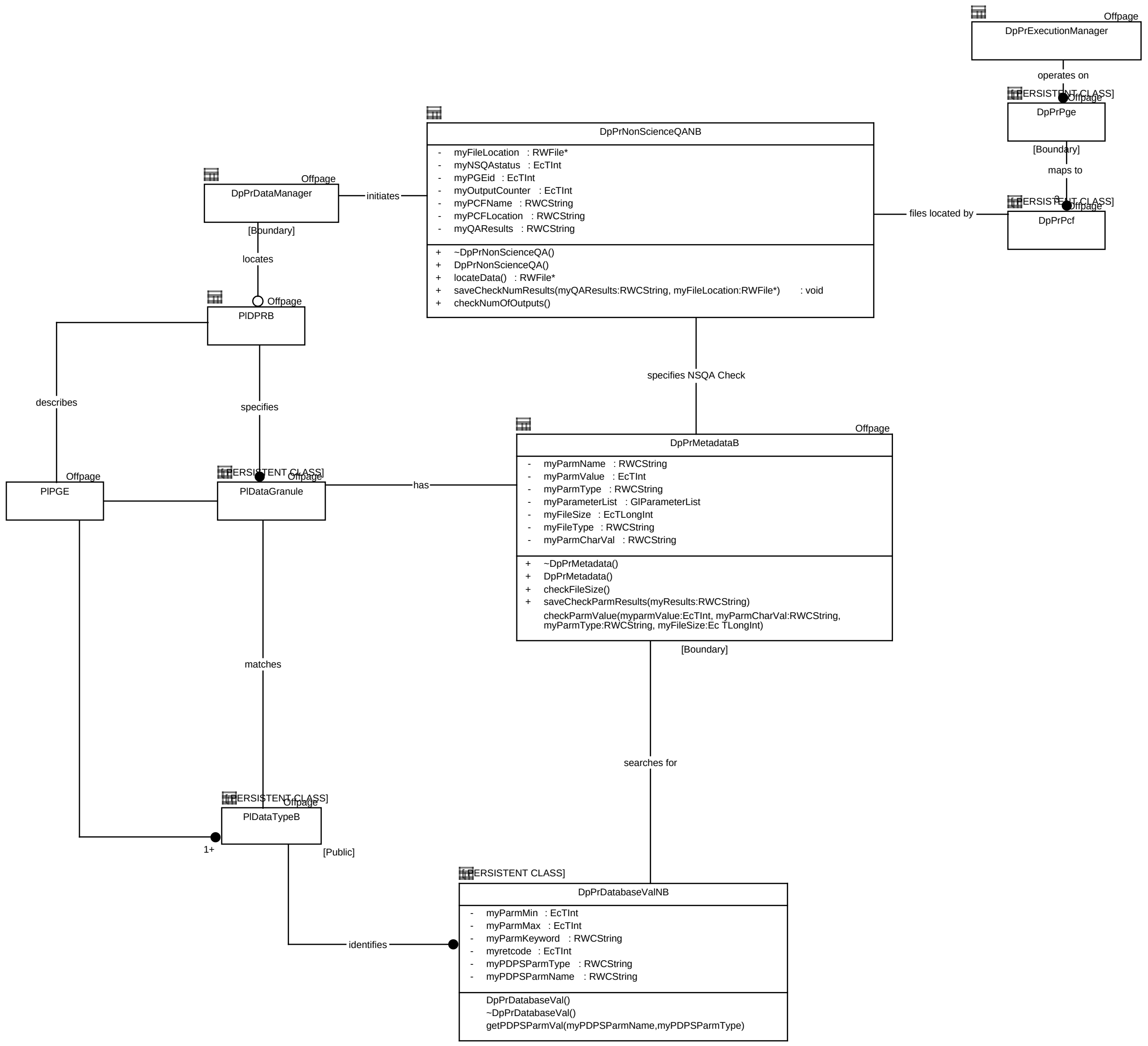


Figure 4.3-7. Metadata QA Object Model

4.3.8 Data Pre-Processing View

The Data Preprocessing View represents the classes used to support the preprocessing of ancillary data which can then be used as inputs to a PGE. Data Preprocessing can be defined as preliminary processing or application of operations on a data set which do not alter or modify its scientific content. Preprocessing includes changes to the format of a data set by reordering the lower level byte structure, reorganization of a data set (ordering data items within and between physical files), preparing additional metadata based on lower level metadata, etc.

Figure 4.3-8, TRMM Data Preprocessing View, represents the overall view of classes for TRMM/ SDPF/FDF data preprocessing. Figure 4.3-9, EDOS Data Preprocessing View, represents an overall view of classes for EDOS data preprocessing. For more detailed views, see the TRMM Onboard Attitude Object Model (Figure 4.3-10), the TRMM Definitive Orbit Object Model (Figure 4.3-11), and the AM-1 Onboard Orbit and Attitude Object Model (Figure 4.3-12).

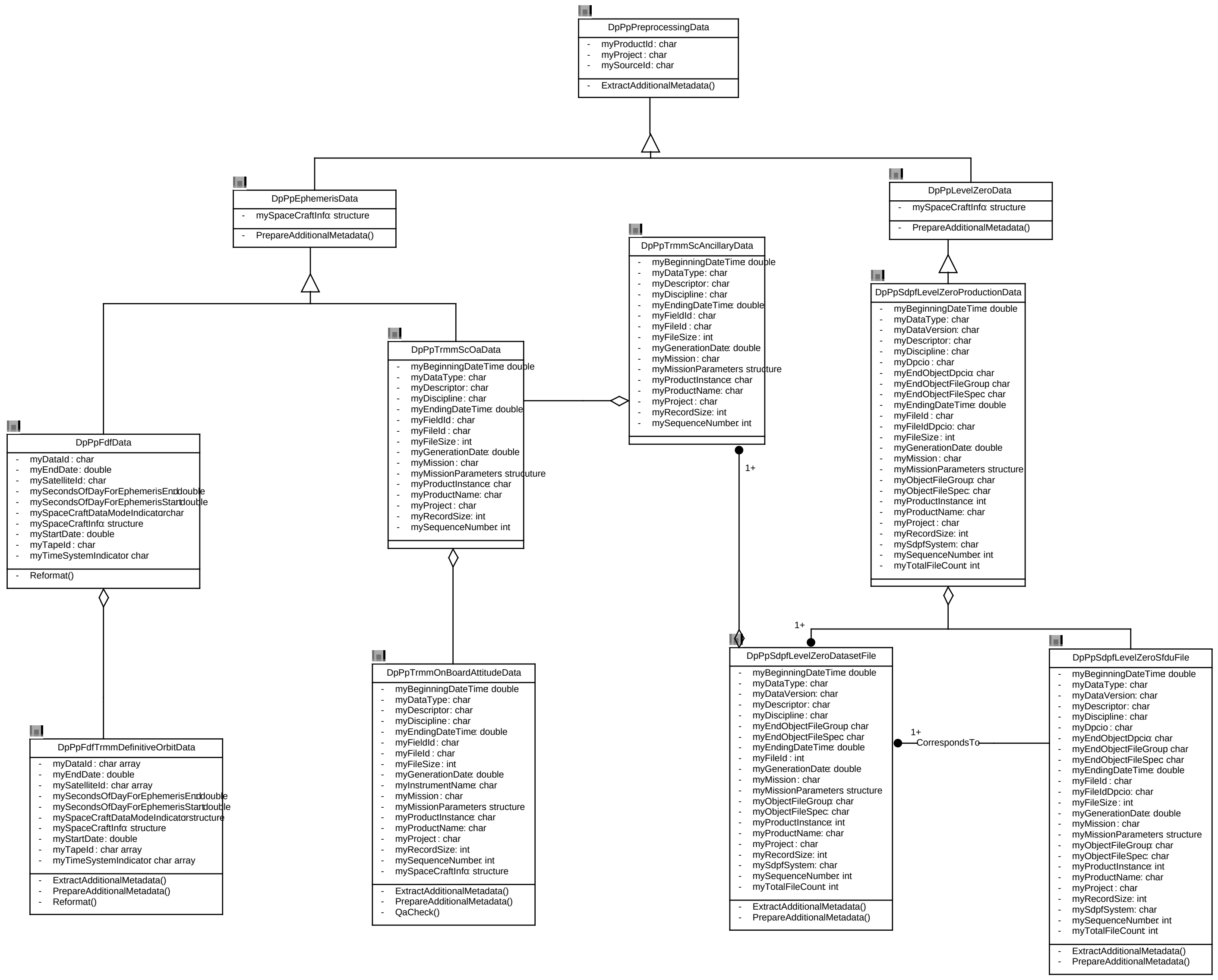
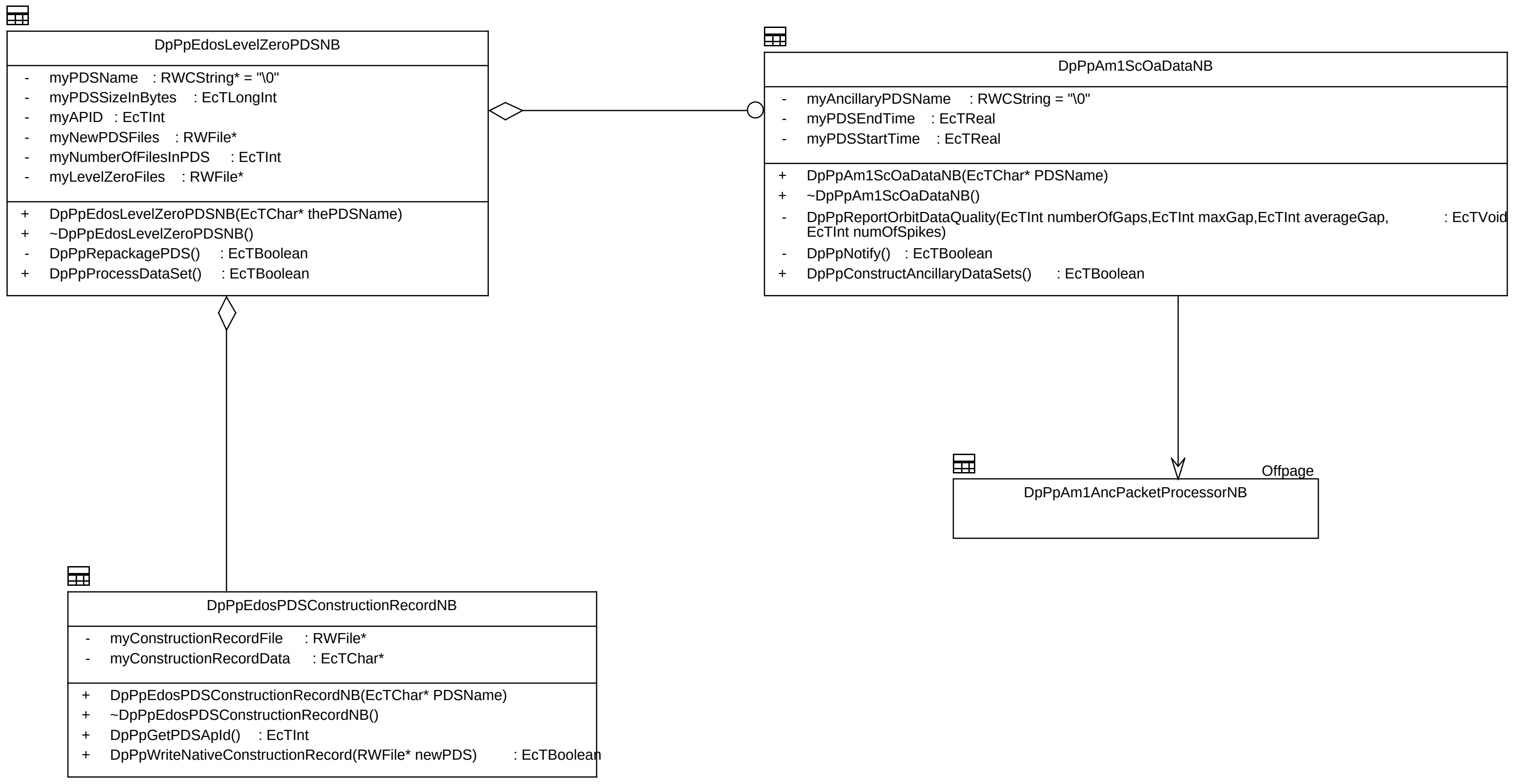


Figure 4.3-8. TRMM Data PreProcessing View



DpPpEdosLevelZeroPDSNB
- myPDSName : RWCString* = "\0"
- myPDSSizeInBytes : EcTLongInt
- myAPID : EcTInt
- myNewPDSFiles : RWFile*
- myNumberOfFilesInPDS : EcTInt
- myLevelZeroFiles : RWFile*
+ DpPpEdosLevelZeroPDSNB(EcTChar* thePDSName)
+ ~DpPpEdosLevelZeroPDSNB()
- DpPpRepackagePDS() : EcTBoolean
+ DpPpProcessDataSet() : EcTBoolean
DpPpAm1ScOaDataNB
- myAncillaryPDSName : RWCString = "\0"
- myPDSEndTime : EcTReal
- myPDSStartTime : EcTReal
+ DpPpAm1ScOaDataNB(EcTChar* PDSName)
+ ~DpPpAm1ScOaDataNB()
- DpPpReportOrbitDataQuality(EcTInt numberOfGaps,EcTInt maxGap,EcTInt averageGap, EcTInt numOfSpikes) : EcTVoid
- DpPpNotify() : EcTBoolean
+ DpPpConstructAncillaryDataSets() : EcTBoolean
Offpage
DpPpAm1AncPacketProcessorNB
DpPpEdosPDSConstructionRecordNB
- myConstructionRecordFile : RWFile*
- myConstructionRecordData : EcTChar*
+ DpPpEdosPDSConstructionRecordNB(EcTChar* PDSName)
+ ~DpPpEdosPDSConstructionRecordNB()
+ DpPpGetPDSApId() : EcTInt
+ DpPpWriteNativeConstructionRecord(RWFile* newPDS) : EcTBoolean

Figure 4.3-9. EDOS Data PreProcessing View

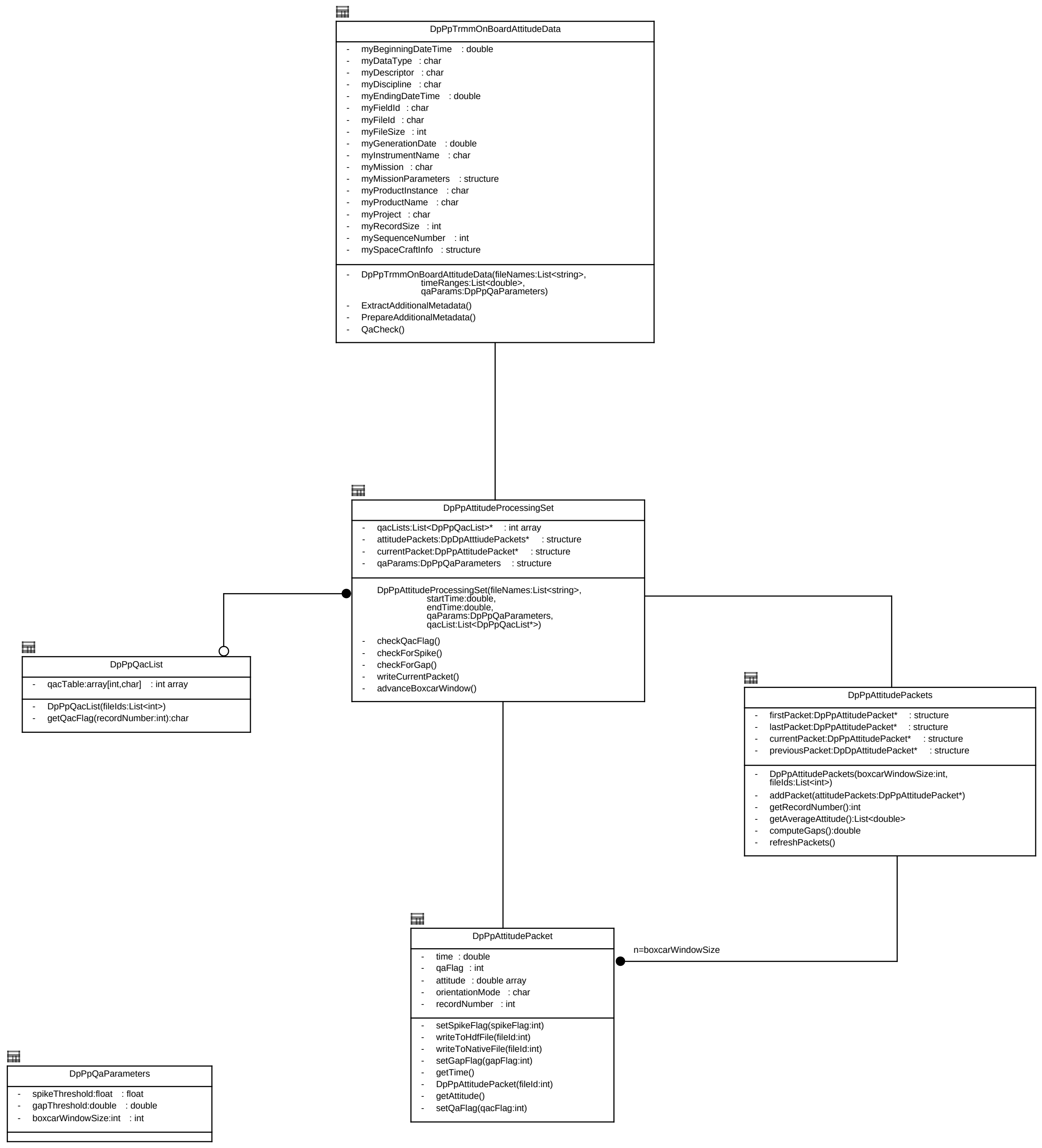
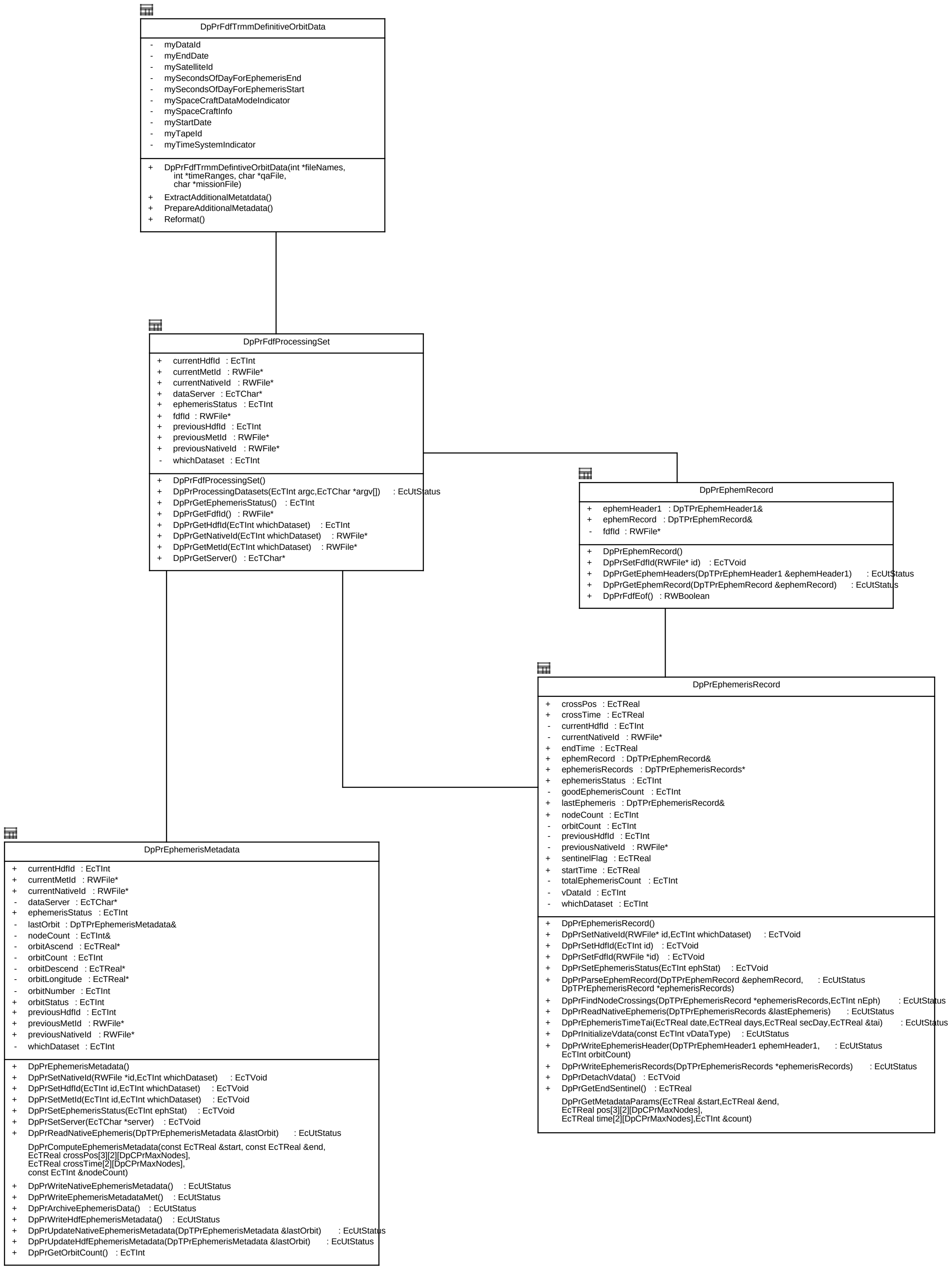

DpPpTrmmOnBoardAttitudeData
- myBeginningDateTime : double
- myDataType : char
- myDescriptor : char
- myDiscipline : char
- myEndingDateTime : double
- myFieldId : char
- myFileId : char
- myFileSize : int
- myGenerationDate : double
- myInstrumentName : char
- myMission : char
- myMissionParameters : structure
- myProductInstance : char
- myProductName : char
- myProject : char
- myRecordSize : int
- mySequenceNumber : int
- mySpaceCraftInfo : structure
- DpPpTrmmOnBoardAttitudeData(fileNames:List<string>, timeRanges:List<double>, qaParams:DpPpQaParameters)
- ExtractAdditionalMetadata()
- PrepareAdditionalMetadata()
- QaCheck()
DpPpAttitudeProcessingSet
- qacLists:List<DpPpQacList>* : int array
- attitudePackets:DpDpAtttiudePackets* : structure
- currentPacket:DpPpAttitudePacket* : structure
- qaParams:DpPpQaParameters : structure
DpPpAttitudeProcessingSet(fileNames:List<string>, startTime:double, endTime:double, qaParams:DpPpQaParameters, qacList:List<DpPpQacList*>)
- checkQacFlag()
- checkForSpike()
- checkForGap()
- writeCurrentPacket()
- advanceBoxcarWindow()
DpPpQacList
- qacTable:array[int,char] : int array
- DpPpQacList(fileIds:List<int>)
- getQacFlag(recordNumber:int):char
DpPpAttitudePackets
- firstPacket:DpPpAttitudePacket* : structure
- lastPacket:DpPpAttitudePacket* : structure
- currentPacket:DpPpAttitudePacket* : structure
- previousPacket:DpDpAttitudePacket* : structure
- DpPpAttitudePackets(boxcarWindowSize:int, fileIds:List<int>)
- addPacket(attitudePackets:DpPpAttitudePacket*)
- getRecordNumber():int
- getAverageAttitude():List<double>
- computeGaps():double
- refreshPackets()
DpPpAttitudePacket
- time : double
- qaFlag : int
- attitude : double array
- orientationMode : char
- recordNumber : int
- setSpikeFlag(spikeFlag:int)
- writeToHdfFile(fileId:int)
- writeToNativeFile(fileId:int)
- setGapFlag(gapFlag:int)
- getTime()
- DpPpAttitudePacket(fileId:int)
- getAttitude()
- setQaFlag(qacFlag:int)
n=boxcarWindowSize
DpPpQaParameters
- spikeThreshold:float : float
- gapThreshold:double : double
- boxcarWindowSize:int : int

Figure 4.3-10. TRMM OnBoard Attitude Object Model



DpPrFdfTrmmDefinitiveOrbitData
- myDataId
- myEndDate
- mySatelliteId
- mySecondsOfDayForEphemerisEnd
- mySecondsOfDayForEphemerisStart
- mySpaceCraftDataModeIndicator
- mySpaceCraftInfo
- myStartDate
- myTapeId
- myTimeSystemIndicator
+ DpPrFdfTrmmDefintiveOrbitData(int *fileNames, int *timeRanges, char *qaFile, char *missionFile)
+ ExtractAdditionalMetatdata()
+ PrepareAdditionalMetadata()
+ Reformat()
DpPrFdfProcessingSet
+ currentHdfId : EcTInt
+ currentMetId : RWFile*
+ currentNativeId : RWFile*
+ dataServer : EcTChar*
+ ephemerisStatus : EcTInt
+ fdfId : RWFile*
+ previousHdfId : EcTInt
+ previousMetId : RWFile*
+ previousNativeId : RWFile*
- whichDataset : EcTInt
+ DpPrFdfProcessingSet()
+ DpPrProcessingDatasets(EcTInt argc,EcTChar *argv[]) : EcUtStatus
+ DpPrGetEphemerisStatus() : EcTInt
+ DpPrGetFdfId() : RWFile*
+ DpPrGetHdfId(EcTInt whichDataset) : EcTInt
+ DpPrGetNativeId(EcTInt whichDataset) : RWFile*
+ DpPrGetMetId(EcTInt whichDataset) : RWFile*
+ DpPrGetServer() : EcTChar*
DpPrEphemRecord
+ ephemHeader1 : DpTPrEphemHeader1&
+ ephemRecord : DpTPrEphemRecord&
- fdfId : RWFile*
+ DpPrEphemRecord()
+ DpPrSetFdfId(RWFile* id) : EcTVoid
+ DpPrGetEphemHeaders(DpTPrEphemHeader1 &ephemHeader1) : EcUtStatus
+ DpPrGetEphemRecord(DpTPrEphemRecord &ephemRecord) : EcUtStatus
+ DpPrFdfEof() : RWBoolean
DpPrEphemerisRecord
+ crossPos : EcTReal
+ crossTime : EcTReal
- currentHdfId : EcTInt
- currentNativeId : RWFile*
+ endTime : EcTReal
+ ephemRecord : DpTPrEphemRecord&
+ ephemerisRecords : DpTPrEphemerisRecords*
+ ephemerisStatus : EcTInt
- goodEphemerisCount : EcTInt
+ lastEphemeris : DpTPrEphemerisRecord&
+ nodeCount : EcTInt
- orbitCount : EcTInt
- previousHdfId : EcTInt
- previousNativeId : RWFile*
+ sentinelFlag : EcTReal
+ startTime : EcTReal
- totalEphemerisCount : EcTInt
- vDataId : EcTInt
- whichDataset : EcTInt
+ DpPrEphemerisRecord()
+ DpPrSetNativeId(RWFile* id,EcTInt whichDataset) : EcTVoid
+ DpPrSetHdfId(EcTInt id) : EcTVoid
+ DpPrSetFdfId(RWFile *id) : EcTVoid
+ DpPrSetEphemerisStatus(EcTInt ephStat) : EcTVoid
+ DpPrParseEphemRecord(DpTPrEphemRecord &ephemRecord, DpTPrEphemerisRecord *ephemerisRecords) : EcUtStatus
+ DpPrFindNodeCrossings(DpTPrEphemerisRecord *ephemerisRecords,EcTInt nEph) : EcUtStatus
+ DpPrReadNativeEphemeris(DpTPrEphemerisRecords &lastEphemeris) : EcUtStatus
+ DpPrEphemerisTimeTai(EcTReal date,EcTReal days,EcTReal secDay,EcTReal &tai) : EcUtStatus
+ DpPrInitializeVdata(const EcTInt vDataType) : EcUtStatus
+ DpPrWriteEphemerisHeader(DpTPrEphemHeader1 ephemHeader1, EcTInt orbitCount) : EcUtStatus
+ DpPrWriteEphemerisRecords(DpTPrEphemerisRecords *ephemerisRecords) : EcUtStatus
+ DpPrDetachVdata() : EcTVoid
+ DpPrGetEndSentinel() : EcTReal
DpPrGetMetadataParams(EcTReal &start,EcTReal &end, EcTReal pos[3][2][DpCPrMaxNodes], EcTReal time[2][DpCPrMaxNodes],EcTInt &count)
DpPrEphemerisMetadata
+ currentHdfId : EcTInt
+ currentMetId : RWFile*
+ currentNativeId : RWFile*
- dataServer : EcTChar*
+ ephemerisStatus : EcTInt
- lastOrbit : DpTPrEphemerisMetadata&
- nodeCount : EcTInt&
- orbitAscend : EcTReal*
- orbitCount : EcTInt
- orbitDescend : EcTReal*
- orbitLongitude : EcTReal*
- orbitNumber : EcTInt
+ orbitStatus : EcTInt
+ previousHdfId : EcTInt
+ previousMetId : RWFile*
+ previousNativeId : RWFile*
- whichDataset : EcTInt
+ DpPrEphemerisMetadata()
+ DpPrSetNativeId(RWFile *id,EcTInt whichDataset) : EcTVoid
+ DpPrSetHdfId(EcTInt id,EcTInt whichDataset) : EcTVoid
+ DpPrSetMetId(EcTInt id,EcTInt whichDataset) : EcTVoid
+ DpPrSetEphemerisStatus(EcTInt ephStat) : EcTVoid
+ DpPrSetServer(EcTChar *server) : EcTVoid
+ DpPrReadNativeEphemeris(DpTPrEphemerisMetadata &lastOrbit) : EcUtStatus
DpPrComputeEphemerisMetadata(const EcTReal &start, const EcTReal &end, EcTReal crossPos[3][2][DpCPrMaxNodes], EcTReal crossTime[2][DpCPrMaxNodes], const EcTInt &nodeCount)
+ DpPrWriteNativeEphemerisMetadata() : EcUtStatus
+ DpPrWriteEphemerisMetadataMet() : EcUtStatus
+ DpPrArchiveEphemerisData() : EcUtStatus
+ DpPrWriteHdfEphemerisMetadata() : EcUtStatus
+ DpPrUpdateNativeEphemerisMetadata(DpTPrEphemerisMetadata &lastOrbit) : EcUtStatus
+ DpPrUpdateHdfEphemerisMetadata(DpTPrEphemerisMetadata &lastOrbit) : EcUtStatus
+ DpPrGetOrbitCount() : EcTInt

Figure 4.3-11. TRMM Definitive Orbit Object

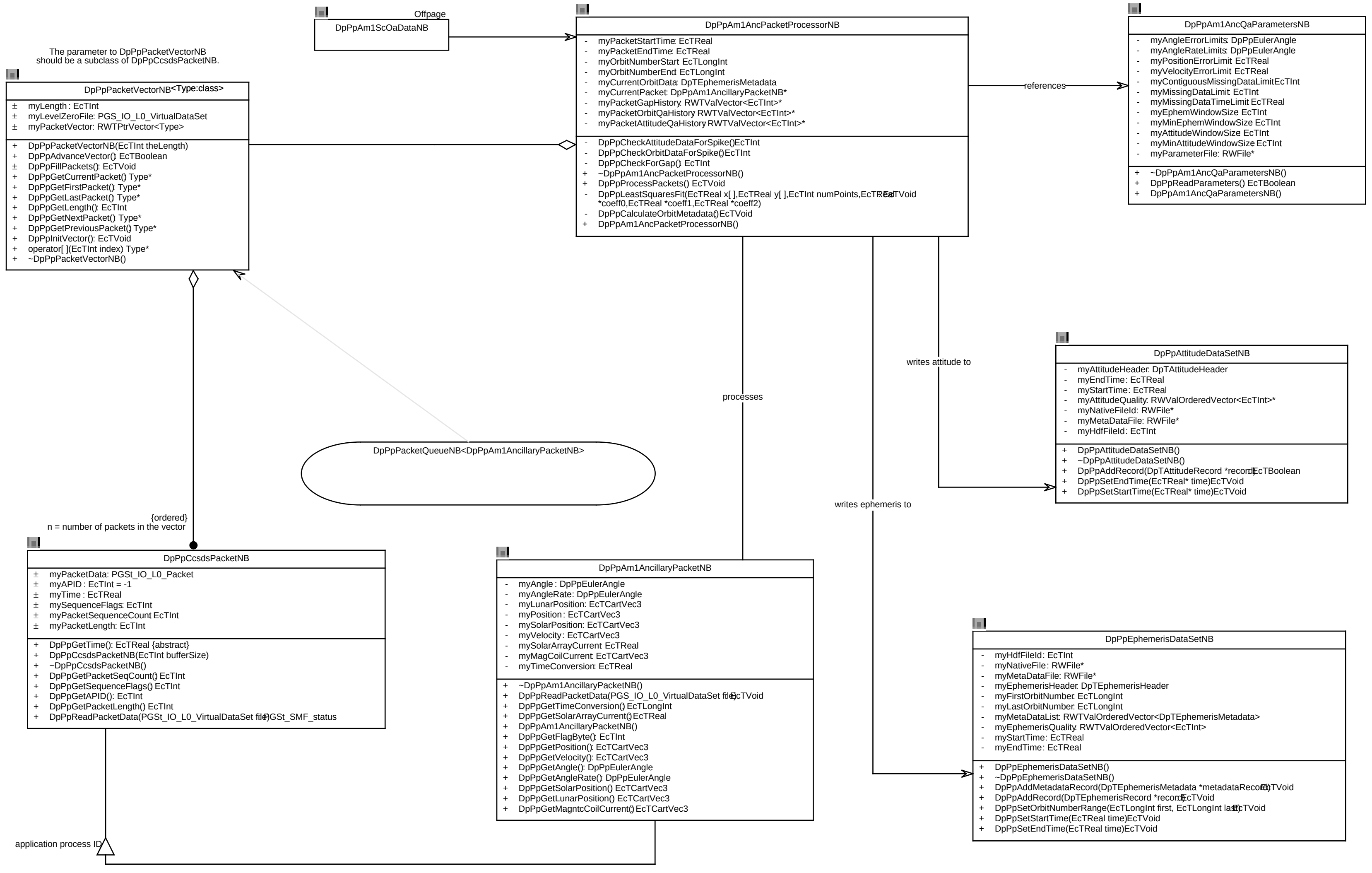
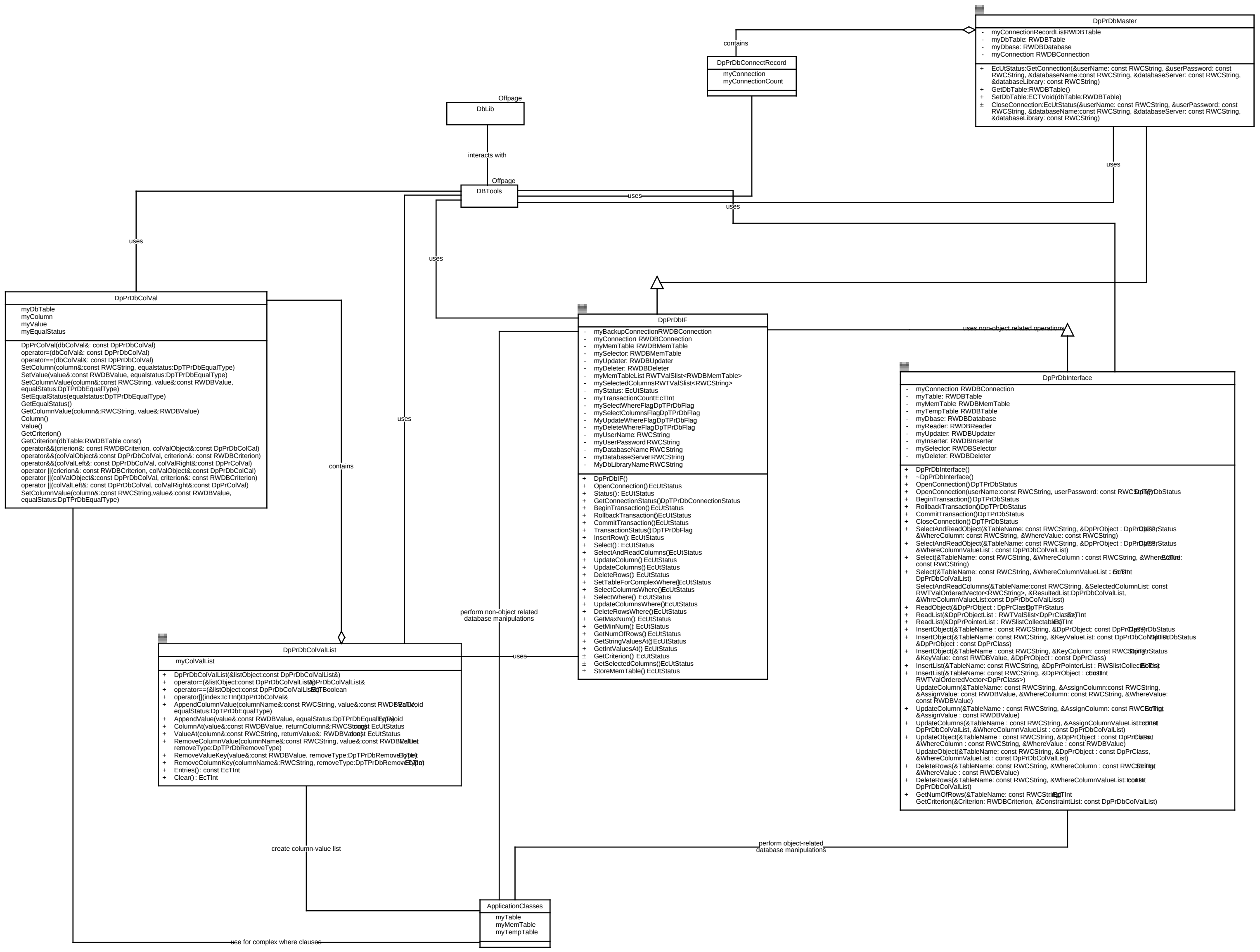

Offpage
DpPpAm1ScOaDataNB
DpPpAm1AncPacketProcessorNB
- myPacketStartTime: EcTReal
- myPacketEndTime: EcTReal
- myOrbitNumberStart: EcTLongInt
- myOrbitNumberEnd: EcTLongInt
- myCurrentOrbitData: DpTEphemerisMetadata
- myCurrentPacket: DpPpAm1AncillaryPacketNB*
- myPacketGapHistory: RWTValVector<EcTInt>*
- myPacketOrbitQaHistory: RWTValVector<EcTInt>*
- myPacketAttitudeQaHistory: RWTValVector<EcTInt>*
- DpPpCheckAttitudeDataForSpike()EcTInt
- DpPpCheckOrbitDataForSpike()EcTInt
- DpPpCheckForGap(): EcTInt
+ ~DpPpAm1AncPacketProcessorNB()
+ DpPpProcessPackets() EcTVoid
- DpPpLeastSquaresFit(EcTReal x[ ],EcTReal y[ ],EcTInt numPoints,EcTReal *coeff0,EcTReal *coeff1,EcTReal *coeff2) EcTVoid
- DpPpCalculateOrbitMetadata()EcTVoid
+ DpPpAm1AncPacketProcessorNB()
references
DpPpAm1AncQaParametersNB
- myAngleErrorLimits: DpPpEulerAngle
- myAngleRateLimits: DpPpEulerAngle
- myPositionErrorLimit: EcTReal
- myVelocityErrorLimit: EcTReal
- myContiguousMissingDataLimitEcTInt
- myMissingDataLimit: EcTInt
- myMissingDataTimeLimit: EcTReal
- myEphemWindowSize: EcTInt
- myMinEphemWindowSize: EcTInt
- myAttitudeWindowSize: EcTInt
- myMinAttitudeWindowSize: EcTInt
- myParameterFile: RWFile*
+ ~DpPpAm1AncQaParametersNB()
+ DpPpReadParameters() EcTBoolean
+ DpPpAm1AncQaParametersNB()
The parameter to DpPpPacketVectorNB should be a subclass of DpPpCcsdsPacketNB.
DpPpPacketVectorNB<Type:class>
± myLength : EcTInt
± myLevelZeroFile: PGS_IO_L0_VirtualDataSet
± myPacketVector: RWTPtrVector<Type>
+ DpPpPacketVectorNB(EcTInt theLength)
+ DpPpAdvanceVector(): EcTBoolean
± DpPpFillPackets(): EcTVoid
+ DpPpGetCurrentPacket() Type*
+ DpPpGetFirstPacket() Type*
+ DpPpGetLastPacket() Type*
+ DpPpGetLength(): EcTInt
+ DpPpGetNextPacket() Type*
+ DpPpGetPreviousPacket() Type*
+ DpPpInitVector(): EcTVoid
+ operator[ ](EcTInt index) Type*
+ ~DpPpPacketVectorNB()
writes attitude to
processes
DpPpAttitudeDataSetNB
- myAttitudeHeader: DpTAttitudeHeader
- myEndTime: EcTReal
- myStartTime: EcTReal
- myAttitudeQuality: RWValOrderedVector<EcTInt>*
- myNativeFileId: RWFile*
- myMetaDataFile: RWFile*
- myHdfFileId: EcTInt
+ DpPpAttitudeDataSetNB()
+ ~DpPpAttitudeDataSetNB()
+ DpPpAddRecord(DpTAttitudeRecord *record) EcTBoolean
+ DpPpSetEndTime(EcTReal* time)EcTVoid
+ DpPpSetStartTime(EcTReal* time)EcTVoid
DpPpPacketQueueNB<DpPpAm1AncillaryPacketNB>
{ordered}
n = number of packets in the vector
writes ephemeris to
DpPpCcsdsPacketNB
± myPacketData: PGSt_IO_L0_Packet
± myAPID : EcTInt = -1
± myTime : EcTReal
± mySequenceFlags: EcTInt
± myPacketSequenceCount EcTInt
± myPacketLength: EcTInt
+ DpPpGetTime(): EcTReal {abstract}
+ DpPpCcsdsPacketNB(EcTInt bufferSize)
+ ~DpPpCcsdsPacketNB()
+ DpPpGetPacketSeqCount() EcTInt
+ DpPpGetSequenceFlags() EcTInt
+ DpPpGetAPID(): EcTInt
+ DpPpGetPacketLength() EcTInt
+ DpPpReadPacketData(PGSt_IO_L0_VirtualDataSet file) PGSt_SMF_status
DpPpAm1AncillaryPacketNB
- myAngle : DpPpEulerAngle
- myAngleRate: DpPpEulerAngle
- myLunarPosition: EcTCartVec3
- myPosition : EcTCartVec3
- mySolarPosition: EcTCartVec3
- myVelocity : EcTCartVec3
- mySolarArrayCurrent: EcTReal
- myMagCoilCurrent: EcTCartVec3
- myTimeConversion: EcTReal
+ ~DpPpAm1AncillaryPacketNB()
+ DpPpReadPacketData(PGS_IO_L0_VirtualDataSet file) EcTVoid
+ DpPpGetTimeConversion() EcTLongInt
+ DpPpGetSolarArrayCurrent() EcTReal
+ DpPpAm1AncillaryPacketNB()
+ DpPpGetFlagByte(): EcTInt
+ DpPpGetPosition(): EcTCartVec3
+ DpPpGetVelocity(): EcTCartVec3
+ DpPpGetAngle(): DpPpEulerAngle
+ DpPpGetAngleRate(): DpPpEulerAngle
+ DpPpGetSolarPosition() EcTCartVec3
+ DpPpGetLunarPosition() EcTCartVec3
+ DpPpGetMagntcCoilCurrent() EcTCartVec3
DpPpEphemerisDataSetNB
- myHdfFileId: EcTInt
- myNativeFile: RWFile*
- myMetaDataFile: RWFile*
- myEphemerisHeader: DpTEphemerisHeader
- myFirstOrbitNumber: EcTLongInt
- myLastOrbitNumber: EcTLongInt
- myMetaDataList: RWTValOrderedVector<DpTEphemerisMetadata>
- myEphemerisQuality: RWTValOrderedVector<EcTInt>
- myStartTime: EcTReal
- myEndTime: EcTReal
+ DpPpEphemerisDataSetNB()
+ ~DpPpEphemerisDataSetNB()
+ DpPpAddMetadataRecord(DpTEphemerisMetadata *metadataRecord) EcTVoid
+ DpPpAddRecord(DpTEphemerisRecord *record) EcTVoid
+ DpPpSetOrbitNumberRange(EcTLongInt first, EcTLongInt last) EcTVoid
+ DpPpSetStartTime(EcTReal time)EcTVoid
+ DpPpSetEndTime(EcTReal time)EcTVoid
application process ID

Figure 4.3-12. AM-1 OnBoard Orbit and Attitude Object Model

4.3.9 Database Interface View

The Database Interface View (Figure 4.3-13) consists of classes used to provide an interface between the application code and the Sybase SQL server housing the PDPS database.



DpPrDbMaster
- myConnectionRecordList: RWDBTable
- myDbTable: RWDBTable
- myDbase: RWDBDatabase
- myConnection: RWDBConnection
+ EcUtStatus:GetConnection(&userName: const RWCString, &userPassword: const RWCString, &databaseName:const RWCString, &databaseServer: const RWCString, &databaseLibrary: const RWCString)
+ GetDbTable:RWDBTable()
+ SetDbTable:ECTVoid(dbTable:RWDBTable)
± CloseConnection:EcUtStatus(&userName: const RWCString, &userPassword: const RWCString, &databaseName:const RWCString, &databaseServer: const RWCString, &databaseLibrary: const RWCString)
contains
DpPrDbConnectRecord
myConnection
myConnectionCount
Offpage
DbLib
interacts with
Offpage
DBTools
uses
uses
uses
uses
uses
DpPrDbColVal
myDbTable
myColumn
myValue
myEqualStatus
DpPrColVal(dbColVal&: const DpPrDbColVal)
operator=(dbColVal&: const DpPrDbColVal)
operator==(dbColVal&: const DpPrDbColVal)
SetColumn(column&:const RWCString, equalstatus:DpTPrDbEqualType)
SetValue(value&:const RWDBValue, equalstatus:DpTPrDbEqualType)
SetColumnValue(column&:const RWCString, value&:const RWDBValue, equalStatus:DpTPrDbEqualType)
SetEqualStatus(equalstatus:DpTPrDbEqualType)
GetEqualStatus()
GetColumnValue(column&:RWCString, value&:RWDBValue)
Column()
Value()
GetCriterion()
GetCriterion(dbTable:RWDBTable const)
operator&&(crierion&: const RWDBCriterion, colValObject&:const DpPrDbColCal)
operator&&(colValObject&:const DpPrDbColVal, criterion&: const RWDBCriterion)
operator&&(colValLeft&: const DpPrDbColVal, colValRight&:const DpPrColVal)
operator ||(crierion&: const RWDBCriterion, colValObject&:const DpPrDbColCal)
operator ||(colValObject&:const DpPrDbColVal, criterion&: const RWDBCriterion)
operator ||(colValLeft&: const DpPrDbColVal, colValRight&:const DpPrColVal)
SetColumnValue(column&:const RWCString,value&:const RWDBValue, equalStatus:DpTPrDbEqualType)
contains
uses
DpPrDbIF
- myBackupConnection RWDBConnection
- myConnection RWDBConnection
- myMemTable RWDBMemTable
- mySelector: RWDBMemTable
- myUpdater: RWDBUpdater
- myDeleter: RWDBDeleter
- myMemTableList RWTValSlist<RWDBMemTable>
- mySelectedColumns RWTValSlist<RWCString>
- myStatus: EcUtStatus
- myTransactionCount EcTInt
- mySelectWhereFlag DpTPrDbFlag
- mySelectColumnsFlag DpTPrDbFlag
- MyUpdateWhereFlag DpTPrDbFlag
- myDeleteWhereFlag DpTPrDbFlag
- myUserName RWCString
- myUserPassword RWCString
- myDatabaseName RWCString
- myDatabaseServer RWCString
- MyDbLibraryName RWCString
+ DpPrDbIF()
+ OpenConnection() EcUtStatus
+ Status(): EcUtStatus
+ GetConnectionStatus() DpTPrDbConnectionStatus
+ BeginTransaction() EcUtStatus
+ RollbackTransaction() EcUtStatus
+ CommitTransaction() EcUtStatus
+ TransactionStatus() DpTPrDbFlag
+ InsertRow(): EcUtStatus
+ Select() : EcUtStatus
+ SelectAndReadColumns() EcUtStatus
+ UpdateColumn() EcUtStatus
+ UpdateColumns() EcUtStatus
+ DeleteRows() EcUtStatus
+ SetTableForComplexWhere() EcUtStatus
+ SelectColumnsWhere() EcUtStatus
+ SelectWhere() EcUtStatus
+ UpdateColumnsWhere() EcUtStatus
+ DeleteRowsWhere() EcUtStatus
+ GetMaxNum() EcUtStatus
+ GetMinNum() EcUtStatus
+ GetNumOfRows() EcUtStatus
+ GetStringValuesAt() EcUtStatus
+ GetIntValuesAt() EcUtStatus
± GetCriterion() EcUtStatus
± GetSelectedColumns() EcUtStatus
± StoreMemTable() EcUtStatus
uses non-object related operations
DpPrDbInterface
- myConnection RWDBConnection
- myTable: RWDBTable
- myMemTable RWDBMemTable
- myTempTable RWDBTable
- myDbase: RWDBDatabase
- myReader: RWDBReader
- myUpdater: RWDBUpdater
- myInserter: RWDBInserter
- mySelector: RWDBSelector
- myDeleter: RWDBDeleter
+ DpPrDbInterface()
+ ~DpPrDbInterface()
+ OpenConnection() DpTPrDbStatus
+ OpenConnection(userName:const RWCString, userPassword: const RWCString) DpTPrDbStatus
+ BeginTransaction() DpTPrDbStatus
+ RollbackTransaction() DpTPrDbStatus
+ CommitTransaction() DpTPrDbStatus
+ CloseConnection() DpTPrDbStatus
+ SelectAndReadObject(&TableName: const RWCString, &DpPrObject : DpPrClass, &WhereColumn: const RWCString, &WhereValue: const RWCString) DpTPrStatus
+ SelectAndReadObject(&TableName: const RWCString, &DpPrObject : DpPrClass, &WhereColumnValueList : const DpPrDbColValList) DpTPrStatus
+ Select(&TableName: const RWCString, &WhereColumn : const RWCString, &WhereValue: const RWCString) EcTInt
+ Select(&TableName: const RWCString, &WhereColumnValueList : const DpPrDbColValList) EcTInt
SelectAndReadColumns(&TableName:const RWCString, &SelectedColumnList: const RWTValOrderedVector<RWCString>, &ResultedList:DpPrDbColValList, &WhreColumnValueList:const DpPrDbColValLisst)
+ ReadObject(&DpPrObject : DpPrClass) DpTPrStatus
+ ReadList(&DpPrObjectList : RWTValSlist<DpPrClass>) EcTInt
+ ReadList(&DpPrPointerList : RWSlistCollectables) EcTInt
+ InsertObject(&TableName : const RWCString, &DpPrObject: const DpPrClass) DpTPrDbStatus
+ InsertObject(&TableName: const RWCString, &KeyValueList: const DpPrDbColValList, &DpPrObject : const DpPrClass) DpTPrDbStatus
+ InsertObject(&TableName : const RWCString, &KeyColumn: const RWCString, &KeyValue: const RWDBValue, &DpPrObject : const DpPrClass) DpTPrStatus
+ InsertList(&TableName: const RWCString, &DpPrPointerList : RWSlistCollectables) EcTInt
+ InsertList(&TableName: const RWCString, &DpPrObject : const RWTValOrderedVector<DpPrClass>) EcTInt
UpdateColumn(&TableName: const RWCString, &AssignColumn:const RWCString, &AssignValue: const RWDBValue, &WhereCoilumn: const RWCString, &WhereValue: const RWDBValue)
+ UpdateColumn(&TableName : const RWCString, &AssignColumn: const RWCString, &AssignValue : const RWDBValue) EcTInt
+ UpdateColumns(&TableName : const RWCString, &AssignColumnValueList : const DpPrDbColValList, &WhereColumnValueList : const DpPrDbColValList) EcTInt
+ UpdateObject(&TableName : const RWCString, &DpPrObject : const DpPrClass, &WhereColumn : const RWCString, &WhereValue : const RWDBValue) EcTInt
UpdateObject(&TableName: const RWCString, &DpPrObject : const DpPrClass, &WhereColumnValueList : const DpPrDbColValList)
+ DeleteRows(&TableName: const RWCString, &WhereColumn : const RWCString, &WhereValue : const RWDBValue) EcTInt
+ DeleteRows(&TableName: const RWCString, &WhereColumnValueList: const DpPrDbColValList) EcTInt
+ GetNumOfRows(&TableName: const RWCString) EcTInt
GetCriterion(&Criterion: RWDBCriterion, &ConstraintList: const DpPrDbColValList)
perform non-object related database manipulations
DpPrDbColValList
myColValList
+ DpPrDbColValList(&listObject:const DpPrDbColValList&)
+ operator=(&listObject:const DpPrDbColValList&) DpPrDbColValList&
+ operator==(&listObject:const DpPrDbColValList&) EcTBoolean
+ operator[](index:IcTInt)DpPrDbColVal&
+ AppendColumnValue(columnName&:const RWCString, value&:const RWDBValue, equalStatus:DpTPrDbEqualType) EcTVoid
+ AppendValue(value&:const RWDBValue, equalStatus:DpTPrDbEqualType) EcTVoid
+ ColumnAt(value&:const RWDBValue, returnColumn&:RWCString) const EcUtStatus
+ ValueAt(column&:const RWCString, returnValue&: RWDBValue) const EcUtStatus
+ RemoveColumnValue(columnName&:const RWCString, value&:const RWDBValue, removeType:DpTPrDbRemoveType) EcTInt
+ RemoveValueKey(value&:const RWDBValue, removeType:DpTPrDbRemoveType) EcTInt
+ RemoveColumnKey(columnName&:RWCString, removeType:DpTPrDbRemoveType) EcTInt
+ Entries() : const EcTInt
+ Clear() : EcTInt
uses
create column-value list
perform object-related database manipulations
ApplicationClasses
myTable
myMemTable
myTempTable
use for complex where clauses

Figure 4.3-13. Database Interface View